Andrey Sadovykh · Dragos Truscan ·
Wissam Mallouli · Ana Rosa Cavalli ·
Cristina Seceleanu ·
Alessandra Bagnato *Editors*

# CyberSecurity in a DevOps Environment

## From Requirements to Monitoring

Springer

# CyberSecurity in a DevOps Environment

Andrey Sadovykh • Dragos Truscan •
Wissam Mallouli • Ana Rosa Cavalli •
Cristina Seceleanu • Alessandra Bagnato
Editors

# CyberSecurity in a DevOps Environment

From Requirements to Monitoring

Springer

*Editors*

Andrey Sadovykh
SOFTEAM
Ivry-sur-Seine, France

Wissam Mallouli
Montimage
Paris, France

Cristina Seceleanu
Mälardalen University
Västerås, Sweden

Dragos Truscan
Information Technologies
Åbo Akademi University
Turku, Finland

Ana Rosa Cavalli
Montimage
Paris, France

Alessandra Bagnato
SOFTEAM
Ivry-sur-Seine, France

ISBN 978-3-031-42211-9 ISBN 978-3-031-42212-6 (eBook)
https://doi.org/10.1007/978-3-031-42212-6

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Paper in this product is recyclable.

# Preface

The idea behind this book came during a group meeting in one of the VeriDevOps European project meetings. We came to a realization that while there exist so many research articles that detail activities related to software security analysis, some more in-depth view in a DevOps cycle including security requirements formalization, verification, and continuous monitoring was needed to present the current state of the art and practice in the field based on the analysis of the literature.

The book aims to provide a comprehensive and systematic overview of the current state of the art and practice in software security analysis, covering topics such as security requirements specification, verification, and continuous monitoring. The book also discusses the challenges and opportunities for future research and practice in this emerging field.

In recent years, security vulnerability reports are omnipresent in many application domains. Skybox Security shows that there were 20,175 new vulnerabilities published in 2021, up from 18,341 in 2020. That's the most vulnerabilities ever reported in a single year, and it's the biggest year-over-year increase since 2018. Vulnerabilities in operational technology jumped to 88%, from 690 in 2020 to 1,295 in 2021. A record 26,448 software security flaws were reported in 2022, with the number of critical vulnerabilities up 59% on 2021 to 4,135 according to analysis by The Stack of Common Vulnerabilities and Exposures (CVEs) data. For instance, in a series of experiments, Tencent's Keen Security Lab exposed critical security vulnerabilities on several car models in their software which could allow a potential attacker to gain access to the car and, for instance, lock the brakes or reprogram some ECUs. Similarly, researchers at CheckPoint Research uncovered a number of security vulnerabilities in the TikTok mobile application allowing attackers to take control of and manipulate accounts and content, including getting access to personal information. In another report, the InfoSec Institute published details about security vulnerabilities related to railway infrastructure components which again will allow hackers to take control of the trains and of the traffic management systems. This alarming trend indicates that the current security practices and tools are not sufficient to cope with the increasing complexity and diversity of software systems. Security vulnerabilities can have severe consequences for the users and organizations that rely on these systems, such as data breaches, identity theft, ransomware attacks, or denial-of-service incidents. Therefore, it is imperative to develop new methods and techniques to identify, prevent, and mitigate security vulnerabilities in software systems.

Elaborated security mechanisms must be properly implemented prior to deployment in order to provide an effective level of protection against intrusion. The number of security scenarios to be ensured explodes. For example, in the embedded software domain, the number of system interactions with the environment that are subject to security attacks is increasing and may result in security vulnerabilities that can cause losses for end users including a drastic increase in the production and maintenance costs, especially if iterations in the development process are long and feedback comes late. In such cases, traditional security verification approaches do not support continuous feedback loops.

Security is not a one-time task but a continuous process that requires constant monitoring and updating. The complexity and diversity of security threats increase with the number and variety of scenarios that need to be covered by the security mechanisms. Therefore, it is important to conduct a thorough analysis of the security requirements and design appropriate solutions that can address them effectively.

As numerous examples show, security is an aspect that has to be addressed holistically from the early phases of the development process and ensured across all phases of the DevOps. Moreover, security quality attributes are often treated after delivery on the code or at the infrastructure level with specific patches, while it is generally agreed that those attributes must be addressed at the design level. DevOps enables fast and frequent software deliveries, which means that artifacts need to be verified quickly and efficiently to keep up with the process. DevOps integrates development and operations to shorten the lead time between a change request and deployment in production. Automation is a crucial technique in modern software development relying on DevOps practices and continuous delivery pipelines. It helps reduce the time between development and normal operations while ensuring high-quality deliverables.

This book is an overview of the latest techniques and tools that can help engineers and developers verify the security requirements of large-scale industrial systems. In addition, it presents novel methods that enable a faster feedback loop for verifying security-related activities. These methods rely on techniques such as automated testing, model checking, static analysis, runtime monitoring, and formal methods. The book aims to provide readers with a practical and theoretical understanding of how to apply these methods in real-world scenarios.

This book covers several advanced topics related to security verification, such as optimizing security verification activities, automatically creating verifiable specifications from security requirements and vulnerabilities, and using these security specifications to verify security properties against design specifications and generate artifacts such as tests or monitors that can be used later in the DevOps process. Security verification is a set of independent procedures that are used to verify that a product, service, or system meets the requirements and specifications and fulfills its intended purpose. This book presents the principles, methods, and tools of security auditing, as well as concrete examples and case studies.

The book is dedicated to a general audience of computer engineers and does not require specific knowledge. It presents the recent updates on the current state of the art and practice in the field based on the analysis of the literature up to date. It is intended for architects, developers, testers, security professionals, tool providers, and consumers who want to define, build, test, and verify secure applications, web services, and industrial systems.

This book consists of three parts, each covering a different aspect of security engineering in the DevOps context. The first part, "Security Requirements," deals with how to specify and analyze security issues in a formal way. The second part, "Prevention at Development Time," offers a practical and industrial perspective on how to design, develop, and verify secure applications. The third part, "Protection at Operations," introduces tools for continuous monitoring of security events and incidents.

## Part I: Security Requirements Engineering

Security requirements engineering is a vital discipline that ensures the development of secure and resilient systems. It involves identifying, analyzing, and specifying security requirements to protect critical assets from potential threats and vulnerabilities.

This part of the book explores the recent state-of-the-art updates in taxonomies, NLP methods applied to security requirements engineering. We delve into the latest advancements and their practical implications in managing security requirements. Moreover, illustrative examples are provided to demonstrate how the methods can be effectively integrated to streamline the security requirements engineering process:

1. "A Taxonomy of Vulnerabilities, Attacks, and Security Solutions in Industrial PLCs." With the ultimate goal of enhancing the security of industrial control systems, this chapter presents a comprehensive taxonomy and mapping study of security vulnerabilities in PLC software. By extracting information from existing studies, the chapter identifies and classifies these vulnerabilities, the corresponding attacks, and the proposed security solutions. This chapter provides invaluable insights for researchers and practitioners involved in mitigating security risks in industrial control systems.
2. "Natural Language Processing with Machine Learning for Security Requirements Analysis – Practical Approaches." This chapter explores NLP's role in analyzing security requirements. Despite their scattered and generic nature, experts extract and detail these requirements using best practices from standards like OWASP ASVS, STIG, or IEC62443. NLP has been applied in requirements engineering (RE) for analysis tasks, although its effectiveness has been uncertain. The chapter outlines the state-of-the-art NLP methods in RE, focusing on

security requirements analysis. Additionally, practical examples illustrate the application of modern transfer learning architectures to important RE tasks.
3. "Security Requirements Formalization with RQCODE." This chapter presents an approach for formalizing the requirements by applying the seamless object-oriented requirements methods and the application of the approach to security requirements. The formalization offers benefits such as improved reuse, a solid premise for verification and validation of security requirements, and reinforcement of system security. The chapter discusses the state-of-the-art requirements formalization and provides illustrative examples.

## Part II: Prevention at Development Time

This part focuses on preventing vulnerabilities during the software development process, by providing first a survey of existing methods for vulnerability detection and response, followed by two novel approaches for security test generation and vulnerability identification in the source code, suitable for industrial systems. The three chapters included in this part are briefly summarized in the following:

1. "Vulnerability Detection and Response: Current Status and New Approaches" presents a taxonomy and mapping study focusing on security vulnerabilities in industrial PLC software. The findings extracted from these studies shed light on the vulnerabilities, corresponding attacks, and proposed solutions. By providing a taxonomy that identifies and classifies these security issues, this chapter offers valuable insights for researchers and practitioners working on mitigating vulnerabilities and attacks in industrial PLC software.
2. "Metamorphic Testing for Verification and Fault Localization in Industrial Control Systems" presents an integrated approach that combines test generation and fault localization using metamorphic testing. Metamorphic relations extracted from system specifications are utilized as derived test oracles to distinguish passed and failed tests for spectrum-based fault localization. The proposed method involves two phases, test generation using metamorphic testing and fault localization for root cause analysis and failure diagnosis, and it has been applied to an industrial PLC system.
3. "Interactive Application Security Testing with Hybrid Fuzzing and Statistical Estimators" introduces an approach that automates the assessment of static analysis results using fuzzing to enable the analysis of large-scale projects. The approach allows one to explore code sections that are typically difficult for traditional fuzzers to reach.

## Part III: Protection at Operations

Protection at operation involves implementing various techniques to enhance security and mitigate risks in real-time environments. Intrusion detection and anomaly detection are crucial components of protection at operations, aimed at identifying unauthorized or abnormal activities that may indicate security threats. These detection mechanisms utilize techniques such as complex event processing, which involves analyzing and correlating events in real time to identify patterns and detect potential threats. Additionally, the concept of explainability plays a vital role in protection at operation by providing insights into the decision-making process of detection algorithms, helping security professionals understand and interpret the results. The combination of intrusion detection, anomaly detection, complex event processing, and explainability contributes to a comprehensive approach to ensure robust protection in operational environments:

1. "CTAM: A Tool for Continuous Threat Analysis and Management." This chapter presents an automated threat analysis toolchain integrated into GitLab DevOps. It enables continuous assessments to threats, monitors progress, and allows advanced analyses. The approach is evaluated on a real-world application to assess threat analysis over time. The chapter concludes with a detailed discussion on using threat modeling in continuous integration.
2. "EARLY – A Tool for Real-Time Security Attack Detection." This chapter introduces an enhanced IDS that detects network attacks early, preventing further harm and downtime. It utilizes deep neural networks trained to extract relevant features from raw network traffic data. The tool is evaluated on two datasets from different domains, showing excellent performance and high overall balanced accuracy.
3. "A Stream-Based Approach to Intrusion Detection." This chapter explores intrusion detection through complex event processing, formalizing pattern matching and runtime monitoring. It introduces a technique to automatically extract relevant elements explaining intrusions, reducing the volume of evidence for manual examination. The approach is evaluated on a proof-of-concept implementation.
4. "Toward Anomaly Detection Using Explainable AI." This chapter presents MMT, a monitoring framework for anomaly detection. It is extended with explainable AI (XAI) capabilities for better understanding AI/ML-based classifications. The chapter includes experimental results using SHAP, LIME, and SHAPASH technologies.

**Acknowledgments** We are grateful to the reviewers who dedicated their time and expertise to evaluate the manuscripts submitted for this book. Their constructive comments and suggestions have helped the authors to enhance the quality and clarity of their chapters: Tanwir Ahmad, Hayretdin Bahsi, Jean-Michel Bruel, Eduard Paul Enoiu, Marcel Kyas, Vinh Hoa La, Ángel Longueira-Romero, Nan Messe, Mikael Ebrahimi Salari, Alessandra Somma, Jüri Vain, Valeria Valdés, and Anis Bouaziz.

This book is largely a result of the VeriDevOps project, which was supported by the Horizon Europe program of the European Commission. We are grateful for the chance to collaborate with our partners for 3 years and to contribute to the advancement of knowledge in this field. The project was a rewarding and enjoyable experience for us, and we hope that the readers will find our research useful and insightful.

| | |
|---|---|
| Paris, France | Andrey Sadovykh |
| Turku, Finland | Dragos Truscan |
| Paris, France | Wissam Mallouli |
| Paris, France | Ana Rosa Cavalli |
| Västerås, Sweden | Cristina Seceleanu |
| Paris, France | Alessandra Bagnato |
| June, 2023 | |

# Contents

# Part I
# Security Requirements Engineering

# Chapter 1
# A Taxonomy of Vulnerabilities, Attacks, and Security Solutions in Industrial PLCs

**Eduard Paul Enoiu, Kejsi Biçoku, Cristina Seceleanu, and Michael Felderer**

**Abstract** In recent years, industrial control systems have been extensively utilized across critical industries, encompassing manufacturing, automation, and power plants. The widespread implementation of these systems within vital infrastructures has escalated the imperative of ensuring their security. This chapter aims to provide a valuable contribution in the form of a taxonomy and a mapping study that addresses security vulnerabilities present in industrial PLC software. The research contains an in-depth analysis of security vulnerabilities, the corresponding exploitative attacks, and the proposed solutions. The primary objective of this chapter is to establish a comprehensive taxonomy that effectively identifies and classifies vulnerabilities, attacks, and solutions pertinent to security in industrial PLCs. Notably, the proposed taxonomy is further demonstrated within the entire DevOps continuum, spanning from the initial design phase to the operational aspect of PLC systems. The outcomes of this research endeavor hold substantial potential in assisting both researchers and practitioners involved in mitigating security vulnerabilities and combatting attacks targeting industrial PLCs.

**Keywords** Programmable Logic Controllers · Security · Taxonomy

E. P. Enoiu (✉) · K. Biçoku · C. Seceleanu
Mälardalen University, Västerås, Sweden
e-mail: eduard.paul.enoiu@mdu.se; kbu20001@student.mdh.se; cristina.seceleanu@mdu.se

M. Felderer
German Aerospace Center (DLR), Köln, Germany

University of Cologne, Köln, Germany
e-mail: michael.felderer@dlr.de

A. Sadovykh et al. (eds.), *CyberSecurity in a DevOps Environment*,
https://doi.org/10.1007/978-3-031-42212-6_1

## 1.1 Introduction

Industrial control systems (ICSs) refer to systems used to monitor, control, automate, and manage essential industrial infrastructures such as oil and natural gas, water, electricity, transportation, and more. These systems play a crucial role in maintaining the stability and efficiency of various critical industries. While control systems offer significant benefits, they also pose several challenges, including critical security vulnerabilities that can have severe consequences [1]. Ensuring the security of industrial control systems (ICSs) is crucial because attacks on these systems can have a direct impact on physical entities under their control, unlike information systems where an attack would only affect the system itself [2]. ICSs consist of multiple components, including programmable logic controllers (PLCs), remote terminal units (RTUs), human-machine interfaces (HMIs), control servers, and more. Each of these components is susceptible to various types of attacks, highlighting the need for a comprehensive and structured approach to identify and mitigate potential vulnerabilities.

While many studies have examined vulnerabilities, attacks, and solutions for industrial PLC software [2–4], none of them has provided a comprehensive overview. This study focuses on classifying security vulnerabilities in industrial PLCs, which are among the most crucial components of ICSs. Additionally, we propose an extensive taxonomy of vulnerabilities, attacks, and solutions related to programmable logic controllers (PLCs).

The security of programmable logic controllers (PLCs) is a critical concern, primarily due to the integration of industrial control systems (ICSs) with external networks and the lack of defensive mechanisms in communication protocols. Unauthorized access by malicious actors could have severe consequences, including loss of life. These aspects are particularly crucial when considering the integration of design and runtime assurance in PLC systems. The DevOps approach aims to bridge the gap between the design and operational phases of PLC systems, reducing costs without compromising security or safety. This is especially relevant for PLC systems, where secure and reliable operation is essential.

We provide a taxonomy of security vulnerabilities, attacks, and solutions in industrial PLC software. By conducting this mapping study and taxonomy development, we aim to achieve the following goals:

1. Address the research gap by focusing on identifying and analyzing security vulnerabilities in industrial PLC systems.
2. Develop a comprehensive classification of the most prevalent security vulnerabilities, attacks, solutions, and preventive measures for industrial PLC software.
3. Create a taxonomy that links these security vulnerabilities, attacks, solutions, and preventive measures for industrial PLC software, providing a structured and systematic approach for identifying and mitigating potential security risks.

## 1.2 Background: Industrial Control Systems

Industrial control systems (ICS) refer to a collection of control systems, networks, controllers, and devices that facilitate the automation of industrial processes across various domains, such as automotive, power plants, water and wastewater, natural gas, and manufacturing [5]. The two most widely used control systems are supervisory control and data acquisition (SCADA) systems and distributed control systems (DCS).

As shown in Fig. 1.1 [6], the components of an ICS include the human machine interface (HMI), remote diagnostics and maintenance utilities, and the control loop, which consists of sensors, actuators, controllers (e.g., PLC), and the controlled process. The sensors deliver variables to the controller, which generates variables for the actuators. The output of the system is considered a process, which can be fully automatic or partially intervened by a human [7]. The HMI enables the configuration of various parameters and provides necessary information in a display. The remote diagnostics and maintenance utilities allow for remote modification and diagnosis of parameters.

Communication between different elements of an ICS relies on communication protocols such as Profibus, Modbus, DNP3, and CIP [8], among others.

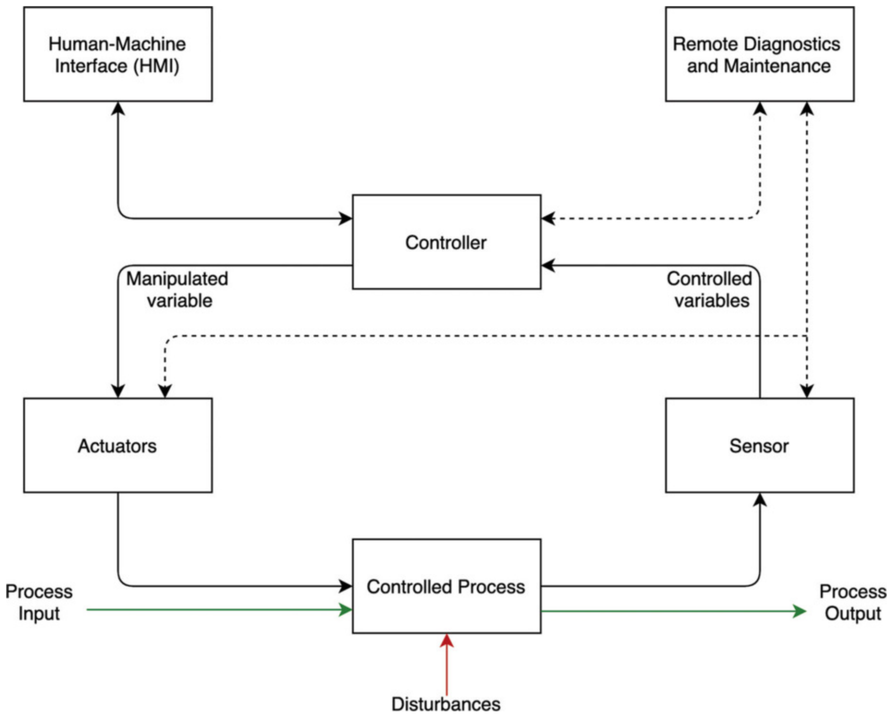


**Fig. 1.1** Industrial control system operation [6]

## 1.3 Related Work

Pan et al. [9] provide an overview of industrial control systems (ICS) and specifically focus on various aspects of PLC security issues and vulnerabilities, such as code, firmware, and network security, Modbus protocol security, and virus vulnerability. The authors analyze these aspects based on previous research and provide available security protection methods.

Sandaruwan et al. [2] emphasize the importance of investigating vulnerabilities in ICSs, with a particular focus on PLCs, which are the most critical components of ICSs. Through various attack vectors, the authors attempt to reveal vulnerabilities that can affect the entire infrastructure. They also provide specific solutions to mitigate the risks associated with these vulnerabilities.

Together, these studies underscore the significance of PLC security in ICSs and highlight the need for identifying and addressing potential vulnerabilities through systematic analysis and effective mitigation strategies.

In their research, Wu et al. [10] highlight a shift in attackers' focus from individual users to industrial control systems (ICSs) and emphasize the significant impact of PLCs' safety on ICSs. They classify security research conducted for PLCs based on function and structure and investigate various aspects such as firmware security, operation, and program security. The authors analyze security measures that focus on defense detention of the PLC program, verification of the PLC firmware's integrity, security encryption of the PLC communication protocol, and formal verification of the PLC code.

On the other hand, Valentine [3] discusses the importance of correct PLC applications and identifies security threats while introducing potential solutions. The research contributions include a taxonomy regarding attacks in ladder logic, ladder logic vulnerabilities, and secure design patterns. However, this taxonomy has a more narrow scope than ours.

Among the studies mentioned, only Valentine's research [3] provides a taxonomy, while others focus on specific aspects of PLC security. Nevertheless, these studies collectively highlight the need for comprehensive security measures for PLCs and the importance of developing taxonomies to identify and address potential vulnerabilities.

While our study systematically derives a taxonomy of security vulnerabilities in industrial PLC software, there are other taxonomies of vulnerabilities, attacks, and security solutions in related domains. For example, Pekaric et al. [11] developed a taxonomy of attack mechanisms in the automotive domain using a similar systematic approach. This highlights the importance of developing structured taxonomies in various domains to enable a better understanding of potential threats and vulnerabilities and to inform the development of effective mitigation strategies.

## 1.4 Method

Our study aims to make a contribution by developing a taxonomy that models and conceptualizes security vulnerabilities in industrial PLC software. A taxonomy is a classification system that defines specific entities based on their characteristic features [12]. By creating a taxonomy, we can systematically identify and classify different types of security vulnerabilities in industrial PLC software, which can inform the development of effective mitigation strategies to address and prevent such vulnerabilities. A mapping study is a crucial component of the taxonomy development process. In this section, we will discuss the design details of both the taxonomy and the mapping study. Section 1.4.1 outlines the taxonomy design, while Sect. 1.4.2 describes the design of the mapping study in detail. By conducting a systematic protocol, we can effectively map the existing literature to identify gaps and overlaps in the field, which helps inform the development of a comprehensive and accurate taxonomy.

### *1.4.1 Taxonomy Protocol*

A taxonomy is a categorization system that aids in differentiating between various research categories for a specific topic. Its primary advantages include providing a systematic overview of the research domain and predicting future research endeavors. The establishment of a taxonomy is supported by conducting a mapping study, which follows the guidelines presented by Usman et al. [13]. The steps involved in executing a mapping study are as follows:

#### 1.4.1.1 Planning

During this phase, we established the taxonomy design in accordance with the following steps:

- *Define software engineering knowledge area:* In [14], the software engineering body of knowledge is categorized into 15 distinct areas. Our study, however, is orthogonal to these areas since it concentrates on software and hardware security.
- *Objective of the taxonomy:* The aim of this taxonomy is to establish relationships between methods for mitigating identified PLC vulnerabilities and the corresponding attacks and solutions. This taxonomy can prove useful to both academics and practitioners, as it enables them to document their research on industrial PLC software in accordance with the established categories.
- *Subject matter:* The subject matter is a more specific definition of the knowledge domain. In this study, the subject matter or units of classification pertain to security vulnerabilities, attacks, and solutions in industrial PLC software.

- *Classification structure type:* When constructing a taxonomy, there are four potential structures that can be employed for the categorization process: paradigm, faceted analysis, tree, and hierarchy. For this study, we have opted to use the faceted analysis structure, which includes multiple facets (e.g., vulnerabilities, attacks, solutions), each with their own attributes.
- *Classification procedure type:* The classification procedures can be classified into two types: qualitative and quantitative. In this particular research, qualitative classification procedures are more appropriate since they rely on nominal scales. These nominal scales are utilized to allocate the subject matter types to the respective dimensions.
- *Identify information sources:* The mapping study described in Sect. 1.4.2 outlines the sources of information from which the data is extracted. Additionally, the study presents the findings obtained from analyzing this information.

#### 1.4.1.2 Identification and Extraction

Once the design of the taxonomy has been planned, the subsequent step involves identifying and extracting the appropriate data. These two phases are carried out through the execution of a mapping study. The following is a general description of each step:

- *Extract terms:* The terminology used for constructing this taxonomy is derived from the data extraction process of the mapping study. Relevant terms and concepts pertaining to vulnerabilities, attacks, and solutions in industrial PLC software are included in a data extraction form.
- *Terminology control:* To prevent any inconsistencies, we ensured terminology control during the data extraction and analysis process.

#### 1.4.1.3 Design

After extracting all relevant data, we must proceed to identify the dimensions, categories, and relationships using the following steps:

- *Identify dimensions:* For this taxonomy, we selected the faceted analysis classification structure, which entails identifying multiple dimensions. The taxonomy comprises the following dimensions: vulnerabilities, attacks, and solutions.
- *Identify categories:* To identify the categories within each dimension, we must utilize either a top-down or bottom-up approach. In this study, we have employed a bottom-up approach, whereby the categories are identified during the data extraction process of the mapping study.
- *Identify relationships:* The dimensions of the taxonomy are interconnected through their association with the security of PLC software. Vulnerabilities and attacks are also linked since attacks exploit vulnerabilities, and similarly, there exists a relationship between attacks and solutions, as solutions aid in

mitigating attacks. A more comprehensive understanding of this interdependence is presented in Sect. 1.6.4.

### *1.4.2 Mapping Study Protocol*

A mapping study is utilized to organize data obtained on a specific subject, identify research trends and gaps, and present findings. To conduct our mapping study, we adhere to the guidelines proposed by Petersen et al. [15] for mapping studies in software engineering. Conducting a mapping study comprises various stages. Our initial step involves defining the overall objective. We then proceed to formulate the search string, select appropriate digital libraries, establish selection criteria, execute the query search, eliminate duplicates, screen papers based on their title and abstract using the selection criteria, screen the full text of papers using the selection criteria, conduct backward snowballing, define the classification framework, extract data from the studies, analyze the extracted data, and present the final results. Figure 1.2 depicts this process.

#### 1.4.2.1 Research Goal

The objective of this research is to establish a classification of the most prominent security vulnerabilities in industrial PLC software, the most prevalent attacks that exploit these vulnerabilities, and potential solutions to enhance the security of industrial PLC software. The findings of this study will serve as a foundation for understanding security vulnerabilities and their underlying causes in industrial PLC software. Thus, our overarching research objective is to establish a taxonomy for categorizing security vulnerabilities, associated attacks, and preventive measures concerning industrial PLC software.

#### 1.4.2.2 Research Questions

This study has formulated the following four research questions that will aid in achieving the defined goal:

- Which categories of security vulnerabilities have been recognized for industrial PLC software?
- Which attacks are the most prevalent in exploiting these security vulnerabilities in industrial PLC software?
- What are the primary solutions or preventive measures for addressing security vulnerabilities in PLC software?

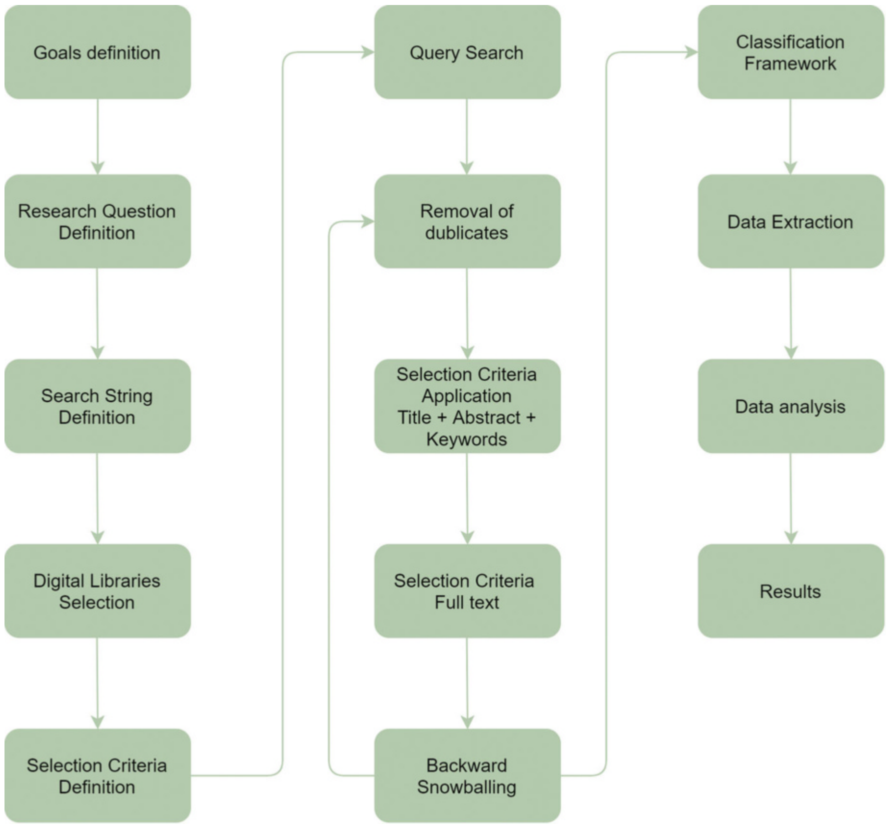

Goals definition
Research Question Definition
Search String Definition
Digital Libraries Selection
Selection Criteria Definition
Query Search
Removal of dublicates
Selection Criteria Application Title + Abstract + Keywords
Selection Criteria Full text
Backward Snowballing
Classification Framework
Data Extraction
Data analysis
Results

**Fig. 1.2** Research method used for the SMS

**Table 1.1** PICO criteria

| | |
|---|---|
| Population | Industrial PLC software |
| Intervention | Security vulnerabilities |
| Comparison | Not applicable |
| Outcomes | Classification of studies based on the vulnerabilities they mention in the context of PLCs |

### 1.4.2.3 Keywords and Search String

The main focus of this study is on security vulnerabilities in industrial PLC software. First, we used the Population, Intervention, Comparison and Outcomes (PICO) criteria to create the search string. The PICO criteria is defined in Table 1.1.
Next, we defined the keywords and their corresponding synonyms and acronyms shown in Table 1.2.

**Table 1.2** Keywords, synonyms and acronyms

| Keywords | Synonyms and acronyms |
|---|---|
| Programmable logic controller | PLC |
| Security | Security |
| Vulnerability | Risk, threat |

We used these keywords, wildcards (i.e., *), and Boolean operators (i.e., AND, OR) and we created the following search string:

(“PLC*” OR “Programmable Logic Controller*”) AND (security) AND (“vulnerabilit*” OR “risk*” OR “threat*”)

#### 1.4.2.4 Digital Libraries

To obtain results for this study, we opted to search two primary digital libraries that are commonly utilized in PLC engineering: IEEE Xplore[1] and Scopus.[2]

#### 1.4.2.5 Selection Criteria

To establish the relevance of studies obtained from our search, we have developed a set of selection criteria, which are classified as either inclusion or exclusion criteria. A study is deemed eligible if it meets all the inclusion criteria and none of the exclusion criteria. Conversely, a study is excluded if it meets at least one exclusion criterion or fails to meet all the inclusion criteria. The selection process is conducted in two stages. In the first stage, papers are assessed based on their title, abstract, and keywords, and in the second stage, the full text of the papers is examined. Inclusion criteria are listed as follows:

- I1. Papers that identify one or more security vulnerabilities in industrial PLC software.
- I2. Papers that are published in conferences or journals.

Exclusion criteria are listed as follows:

- E1. Papers which are duplicative or outdated versions of prior papers.
- E2. Papers which are not peer-reviewed.
- E3. Papers which are secondary or tertiary studies.
- E4. Papers which are not accessible in English.
- E5. Papers which are not accessible in full text.

[1] https://ieeexplore.ieee.org
[2] https://www.scopus.com/

- E6. Papers which address security concerns in ICS but not explicitly in relation to PLCs.

#### 1.4.2.6 Query Search

After the search string and digital libraries are defined, the search string is used in those digital libraries to get the results from the automatic search. The following search strings are the specific ones for each library:

**IEEE Xplore:** (“PLC*” OR“Programmable Logic Controller*”) AND (security) AND (“vulnerabilit*” OR “risk*” OR “threat*”)

**Scopus:** (“PLC*”OR“Programmable Logic Controller*”) AND (security) AND (“vulnerabilit* ” OR “risk*”OR“threat*”) AND (LIMIT-TO (OA , “all”)) AND (LIMIT-TO(DOCTYPE, “ar”) OR LIMIT-TO (DOCTYPE , “cp”)) AND (LIMIT-TO(SUBJAREA ,“COMP”) OR LIMIT-TO (SUBJAREA , “ENGI”)) AND (LIMIT-TO(LANGUAGE , “English”)) AND (LIMIT-TO(PUBSTAGE ,“final”)) AND (LIMIT-TO(SRCTYPE , “j”) OR LIMIT-TO (SRCTYPE , “p”))

All the studies obtained through automated searches of the libraries have been exported and subsequently examined in two distinct spreadsheets.

#### 1.4.2.7 Selection Criteria Application

Following the elimination of duplicates, it becomes imperative to assess the pertinence of the remaining papers. Their relevancy is determined based on the selection criteria outlined in Sect. 1.4.2.5. The process of selecting studies involves two steps: first, applying the selection criteria to the title, abstract, and keywords, and, second, applying the selection criteria to the full text.

#### 1.4.2.8 Classification, Extraction, and Analysis

Our inclusion of domain-specific information pertains specifically to topics about industrial PLC software. The classification framework comprises a vulnerabilities category, which is populated by extracting information on security vulnerabilities in industrial PLC software from the papers. Additionally, the framework encompasses both attacks and solutions categories.

During the analysis phase of data collection, the data extracted from the primary studies is utilized to construct the taxonomy. Initially, the data for each category of the classification scheme is scrutinized, and the quantity of studies in a particular category is determined. Nevertheless, this data is solely applicable to each individual category, and it is crucial to examine how the categories are interlinked.

## 1.5 Search Results

In this section, we present the number of papers that remain for each stage of the study, starting from the digital library search to the final set of papers, after outlining all the required steps and components. Figure 1.3 provides a summary of

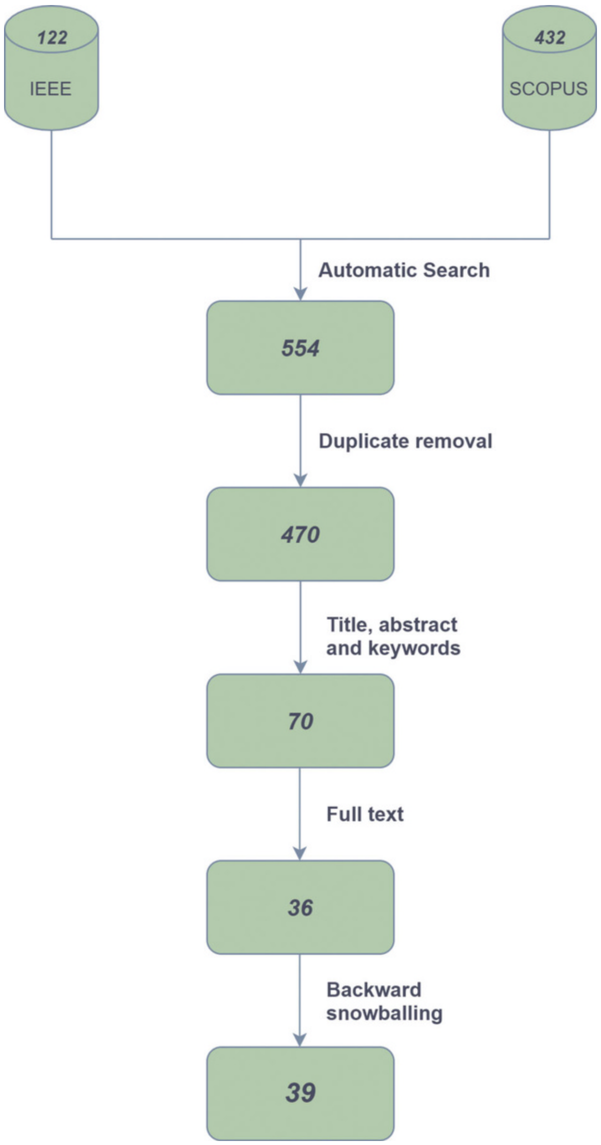


**Fig. 1.3** Search and selection results

**Table 1.3** Search Result

| Digital library | Search results |
|---|---|
| IEEE XPlore | 122 |
| SCOPUS | 432 |
| Total | 554 |

this process, and a thorough explanation of the process is provided subsequently. As outlined in Sect. 1.4.2.6, the automatic search process involved the execution of the search string in two distinct digital libraries. Table 1.3 displays the results of our search, indicating the number of papers retrieved from both IEEE Xplore and SCOPUS. Our search criteria matched 122 papers in IEEE Xplore and 432 papers in SCOPUS. By combining the results from both sources, we obtained a total of 554 papers.

We executed the removal of duplicates, which led to the elimination of 84 papers. Initially, we applied the selection criteria to the papers' titles, abstracts, and keywords, resulting in the exclusion of 400 papers. Most of the excluded studies either did not meet I1 or met E6. In the subsequent stage, we applied the selection criteria to the full text, excluded 34 studies, and retained 36 studies. The majority of the papers excluded in this stage either met E5 or E6. Backward snowballing was conducted to add more papers to the final set to extract more relevant results. From the 36 papers, we collected 578 studies. From these 578 studies, 493 were excluded based on the title and keywords, and 67 were excluded based on the abstract. From applying the selection criteria to the full text, 15 studies were excluded. In total, from the backward snowballing process, we obtained three additional papers. Since the snowballing process was the last process before the classification framework definition, our final set of papers contains 39 studies that can be used to build this taxonomy.

## 1.6 Taxonomy Results

In this section, we present a summary of the findings from our data analysis. We not only identify the different categories of PLC vulnerabilities but also explore the interconnections among them, as well as between vulnerabilities, attacks, and solutions.

### *1.6.1 Security Vulnerabilities*

During the data extraction process, ten primary categories of vulnerabilities were identified. While some of these categories are interconnected (e.g., lack of encryption or authentication may stem from protocols' lack of security when used on the

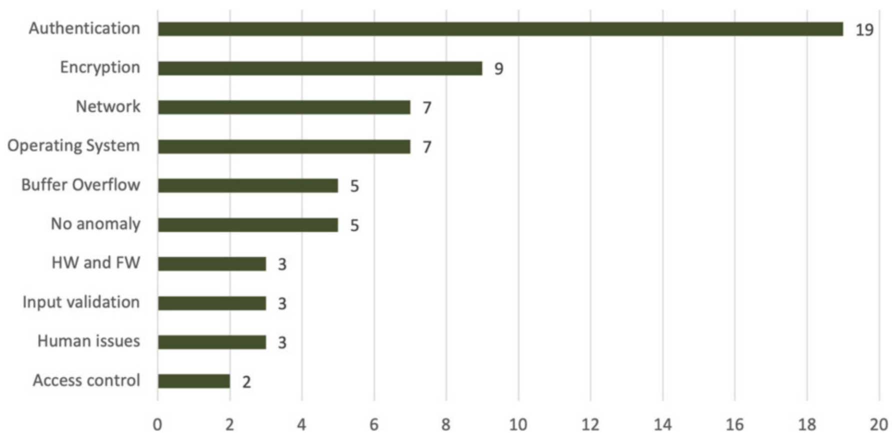


**Fig. 1.4** Vulnerabilities in PLC systems

Internet), we classified them separately as authentication, encryption, and network to avoid any potential bias. Additionally, this categorization allows us to see how many studies address networks generally versus specific areas. Figure 1.4 provides a visual summary of the results.

The category with the highest number of publications is related to authentication, with 19 studies dedicated to this topic. This category primarily discusses the vulnerabilities associated with hard-coded passwords, lack of two-factor authentication mechanisms, and absence of passwords altogether.

The categories that receive the most attention in terms of publications are authentication and encryption. This is mainly because communication protocols used in ICS lack security features. Studies such as S4, S8, and S9 focus on this issue. For instance, the Modbus communication protocol, which is one of the oldest and most commonly used protocols in ICS, does not offer any authentication or encryption. As a result, an attacker can easily obtain the necessary information for a Modbus session to be valid (e.g., function code and address) from a network sniffer like Wireshark and launch an attack. Moreover, Modbus messages are not encrypted and are transmitted in plain text. This issue is not limited to Modbus, as other widely used protocols such as Profibus, Profinet, EtherCAT, and more also lack encryption.

The absence of anomaly detection mechanisms is highlighted as a vulnerability in five studies, as they emphasize that the inability to detect unusual behavior of PLCs in real time can result in disastrous consequences. This vulnerability is not only perilous because of the critical systems that are governed by PLCs but also because it is challenging to identify the root cause of the attack while it spreads throughout the system [16].

Operating system security is identified as a vulnerability in four studies that focus on industrial PLC software. Similar to other operating systems, the operating systems used in these PLCs also have vulnerabilities that can be exploited by attackers. For instance, some of the vulnerabilities mentioned in these studies include the Microsoft Windows server service vulnerability (MS-08-067), the print spooler vulnerability (MS10-061), and the Microsoft Windows.LNK/.PIF vulnerability (MS10-046), as noted in S33.

Four studies mention the buffer overflow vulnerability, which is associated with memory corruption. When this vulnerability is exploited, the attacker can modify program execution [17]. Additionally, this vulnerability is closely connected to another vulnerability discussed in three studies, namely, the lack of input validation. By validating input, the program will only accept values within an acceptable range, such as input character length, and prevent buffer overflow [17, 18].

The three studies that discuss the lack of information about hardware and firmware present a conflicting situation. On the one hand, PLC vendors require security research and measures to be taken against possible attacks. However, since the hardware and firmware they use are proprietary and not publicly available, conducting research to obtain reliable results on how to mitigate the possibility of attacks is nearly impossible [17, 19]. Therefore, this vulnerability is primarily due to the vendors' choices rather than the system itself.

Human issues are mentioned as a vulnerability in three studies (S34-S36). This category encompasses inexperienced developers, naive users who might unintentionally make the system vulnerable to attacks, or personnel who may intentionally try to attack the system. Access control, on the other hand, is the least frequently mentioned vulnerability category, with only two papers discussing it. S1 and S43 focus on the importance of providing users with information based on their needs and granting privileged access only to authorized users. Refer to Table 1.4 for the papers in each category.

**Table 1.4** Vulnerabilities and studies

| Vulnerabilities | Studies |
|---|---|
| Encryption | S1, S4, S7, S12, S20, S23, S24, S30, S34 |
| Authentication | S1–S4, S6–S9, S11, S13, S21–S23, S27, S30–S32, S34, S37 |
| No anomaly detection | S12, S17, S19–S21 |
| Access control | S1, S27 |
| Human issues | S34–S36 |
| Operating System | S15, S19, S33, S34, S37–S39 |
| Buffer Overflow | S7, S19, S25–S27 |
| Input validation | S11, S14, S27 |
| Lack of information about hardware and firmware | S18, S24, S27 |
| Network | S5, S10, S15, S16, S27–S29 |

The primary outcomes from the gathered data on security vulnerabilities in industrial PLC software are as follows:

*The research on security vulnerabilities in industrial PLC software has a significant focus on issues related to authentication, encryption, networks, and operating systems.*

### 1.6.2 Attacks

The graphical representation of the number and type of attacks in industrial PLC software can be seen in Fig. 1.5.

The most commonly mentioned attack that takes advantage of the security weaknesses in industrial PLC software is denial of service (DoS), discussed in 11 studies. A system is designed to handle a certain amount of traffic, and if more traffic is directed to a particular address with the intention of rendering the system unavailable to users, it is considered a DoS attack [20]. In PLC industrial settings, availability is a critical attribute, making the impact of this attack significant.

Stuxnet is a worm that gained notoriety for infecting PLCs and taking control of the gas centrifuges in Iran, causing them to spin at high speeds until they burned out. The worm searched for an industrial automated software called SIMATIC Step 7 used by PLCs on infected PCs. Once it found this software, it would also find a PLC and inject malicious code into it. The worm was designed to send false data to cover its tracks and avoid detection by controllers, making it difficult to detect the attack [21, 22].

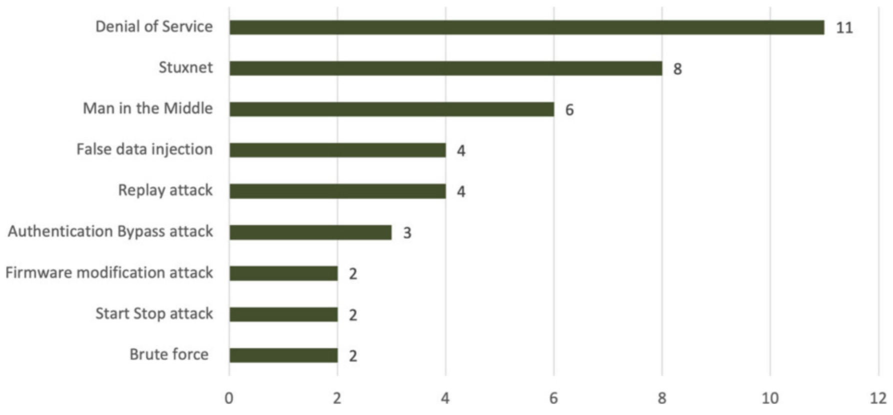


**Fig. 1.5** Attacks

The man-in-the-middle attack (MITM), which is mentioned in six studies, is another well-known attack. In this type of attack, the attacker positions themselves in between an HMI and a PLC and intercepts all the information that the HMI is attempting to send to the PLC. This allows the attacker to modify the data they receive from the HMI and send it to the PLC, without the PLC being aware that the data has been tampered with. Additionally, the attacker can also observe or attempt to block the traffic [20].

A similar attack is the replay attack mentioned in four studies (i.e., S4, S16, S27, S30). In this attack, the attacker also gets a copy of the information exchanged between the two hosts and can later use it by sending duplicate information.

The false data injection attack is mentioned in four studies (i.e., S3, S6, S11, S17). The PLC receives information from sensors, and during the false data injection attack, this information is manipulated. When this happens, the PLC will output commands according to the false data injected and not according to the real measurements from the sensor. That could lead to damaging incidents.

Two studies (S8 and S30) mention the brute force attack, which involves an algorithm that attempts all possible combinations for a specific password until the correct password is identified and the necessary credentials are obtained to access the system. As a result, it is recommended to use passwords with a minimum number of characters, as well as a combination of lowercase and uppercase letters, numbers, and special characters in most applications. A larger alphabet results in a larger number of possible combinations that must be tried to discover the correct password. S8 describes a similar attack called the dictionary attack, in which the attacker uses a list of previously used or common passwords instead of trying all possible character combinations.

The authentication bypass attack is mentioned in two studies and takes advantage of the lack of security in the protocol. An attacker can obtain an authentication packet from a validated user in the system and use it to authenticate themselves [23]. All other attacks are only mentioned once in the reviewed studies. Table 1.5 presents the papers for each attack category.

*The prevalent types of attacks that exploit security vulnerabilities in industrial PLC software include denial of service, Stuxnet, man-in-the-middle, replay attacks, and false data injection attacks.*

### *1.6.3 Security Solutions*

Figure 1.6 provides a visual depiction of the proposed solutions for mitigating vulnerabilities and attacks in industrial PLC software. Of the 39 studies examined, only 19 (or approximately half) discuss potential solutions for mitigating attacks or

**Table 1.5** Attacks and studies

| Attacks | Studies |
|---|---|
| Denial of service | S1, S5, S11, S12, S17, S21, S22, S14, S25, S28, S36 |
| Stuxnet | S11, S24, S28, S29, S31, S33, S36, S39 |
| Man-in-the-middle | S12, S13, S16, S22, S24, S30 |
| Replay attack | S4, S16, S27, S30 |
| False data injection | S3, S6, S11, S17 |
| Brute force | S8, S30 |
| Authentication bypass attacks | S5, S30, S34 |
| Start stop attack | S20, S34 |
| Dictionary attack | S8 |
| Phishing | S8 |
| PLC-PC worm | S10 |
| SQL injection | S11 |
| Data execution attack | S13 |
| Control logic attack | S13 |
| Stealth command modification attack | S16 |
| Interception attack | S17 |
| Maroochi attack | S24 |
| Duqu | S29 |
| Havex | S24 |
| Firmware modification attack | S38, S39 |

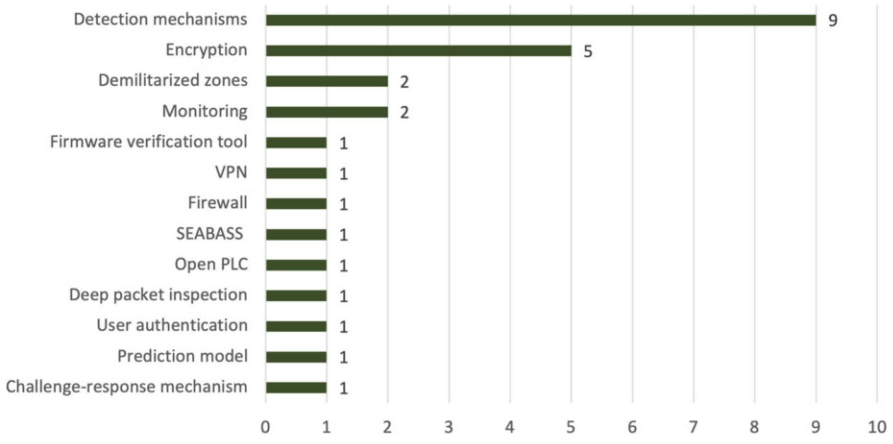


**Fig. 1.6** Solutions for vulnerabilities

securing vulnerable system components. In total, we identified 13 distinct solutions, with nine mentioned only once.

The solutions most frequently cited in the PLC literature involve detection mechanisms. These mechanisms are designed to identify patterns or events that diverge from typical system behavior. By monitoring system behavior, changes that indicate a potential attack can be detected and reported to a control center [24, 25]. Intrusion detection is mentioned in four of the nine papers (S2, S13, S17, S30), while anomaly detection is mentioned in three of the nine papers (S7, S26, S29). Data tampering detection is mentioned in one paper (S6), as is attack model detection (S15). While these mechanisms are all forms of detection, we have grouped them together. It is important to note that these detection mechanisms only serve to identify attacks and do not provide a means of mitigating them.

One potential approach is encryption. Many commonly used communication protocols transmit messages in plain text, leaving them vulnerable to exploitation by attackers. To address this issue, five papers recommend encrypting the content of messages to ensure confidentiality. Users can encrypt their messages, which can then be decrypted by the intended recipients with the proper decryption key [21]. The underlying concept is to incorporate encryption mechanisms into communication protocols or replace existing protocols with ones that support encryption.

Another potential solution is the implementation of demilitarized zones. As a significant portion of cyber vulnerabilities and attacks stem from Internet connections, the goal of demilitarized zones is to isolate the primary network from the Internet, which is generally deemed insecure. In the event of an attack, this setup prevents the attacker from accessing the primary network, limiting their access to only the untrusted segment [2, 26]. This approach also allows for partitioning of the network into multiple zones, each with its own protective layer. If one zone becomes infected, it is unable to spread the infection to other zones.

As the remaining solutions are mentioned only once, we do not offer a detailed explanation of them. However, for interested readers, we have included the papers that discuss these solutions in our data extraction form for further reference. The relevant papers for each category are listed in Table 1.6.

**Table 1.6** Solutions and studies

| Solutions | Studies |
|---|---|
| Detection mechanisms | S3, S6, S7, S13, S15, S17, S26, S29, S30 |
| Encryption | S16, S17, S24, S30, S31 |
| Monitoring | S20, S32 |
| Demilitarized zones | S30, S32 |
| Challenge-response mechanism | S8 |
| Prediction model | S10 |
| User authentication | S13 |
| Deep packet inspection | S13 |
| Open PLC | S18 |
| SEABASS | S22 |
| Firewall | S32 |
| VPN | S32 |
| Firmware verification tool | S38 |

*Various techniques have been utilized in research to provide solutions for security vulnerabilities in PLC software, including the implementation of detection mechanisms and encryption.*

### 1.6.4 A Taxonomy for PLC-Based Vulnerabilities, Attacks, and Security Solutions

The data collected in the study was utilized to develop a taxonomy for organizing the vulnerabilities, attacks, and solutions in PLCs. Figure 1.7 displays a faceted taxonomy that encompasses all of these elements. The root of the taxonomy is the PLC, while the taxonomy's dimensions consist of vulnerabilities, attacks, and solutions. Each dimension includes multiple categories, with ten categories identified for vulnerabilities, 20 for attacks, and 13 for solutions.

Since we are connecting the vulnerabilities, attacks, and solutions for industrial PLC software into a single taxonomy, in Fig. 1.8 we describe the relationship between these three dimensions. As mentioned throughout this study, vulnerabilities are weaknesses of the PLC system. They pose a risk to the PLC system as they *lead to a* different attack that exploits these vulnerabilities. These attacks are *resolved by* different solution mechanisms that *mitigate* the corresponding vulnerabilities.

Table 1.7 presents a mapping of the security vulnerabilities found in the previous study to their corresponding attacks and solutions. It can be observed that each vulnerability can result in multiple attacks and each attack can have multiple solutions. The majority of the reported attacks were related to PLC network security, indicating that more research is needed in this area to develop effective PLC security mechanisms. This tabular taxonomy provides a comprehensive framework that includes the dimensions of vulnerabilities, attacks, and solutions. The vulnerabilities dimension is divided into 10 categories, the attacks dimension into 20 categories, and the solutions dimension into 13 categories.

Figure 1.9 illustrates the relationship between vulnerabilities, attacks, and solutions, focusing only on attacks that exploit a vulnerability and have a possible solution. Attacks serve as the connecting point between vulnerabilities and solutions. For instance, with respect to FDIA, authentication is the most commonly identified vulnerability, whereas detection mechanisms and encryption are the most frequently employed solutions. Regarding the MITM attack, the lack of encryption is the predominant vulnerability, while encryption mechanisms are the most common solutions. Similarly, for the replay attack, the primary vulnerabilities are the lack of authentication, encryption, and network issues, while encryption mechanisms are the main solutions.

The taxonomy developed in this study provides several contributions. Firstly, the process used to create this taxonomy can serve as a model for other researchers

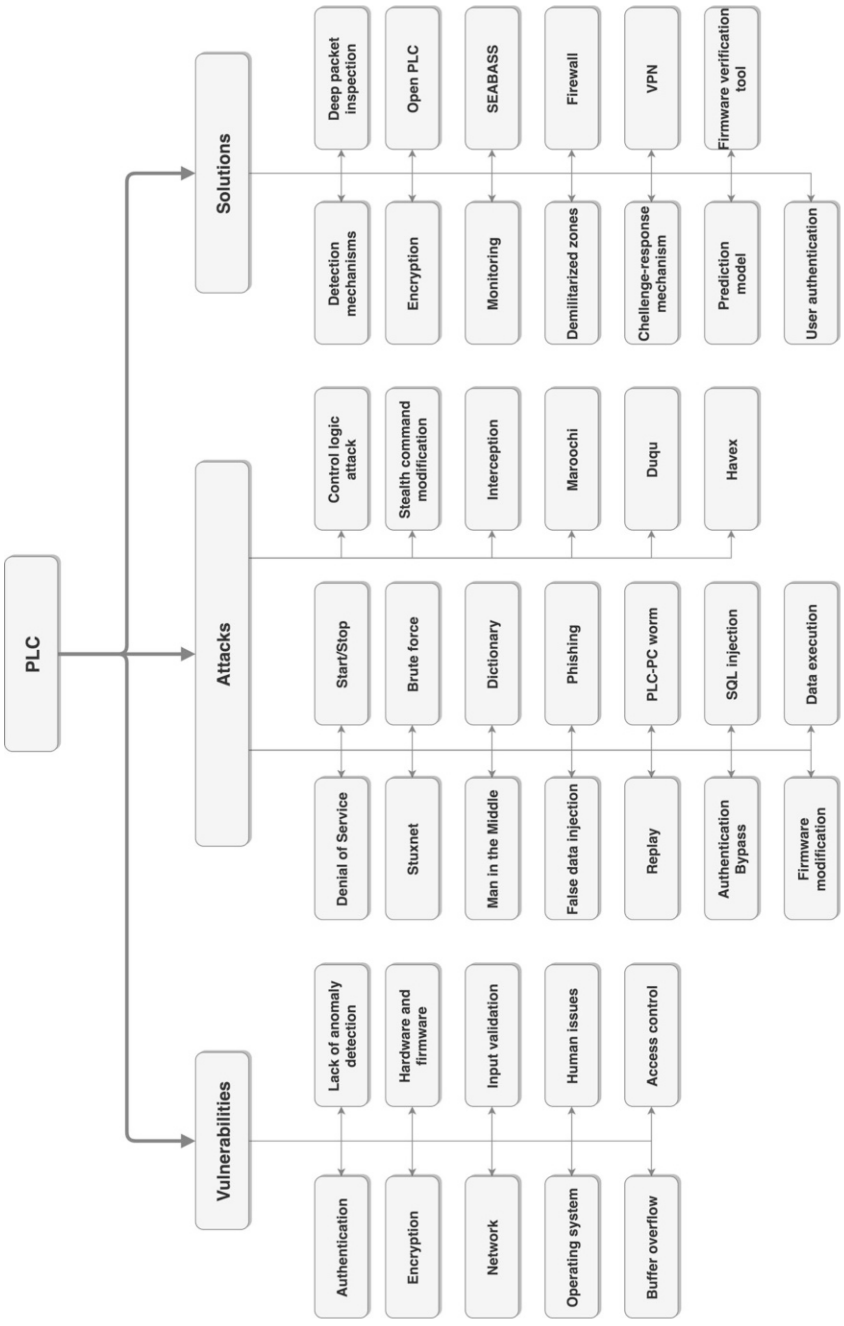


**Fig. 1.7** Security taxonomy for PLCs

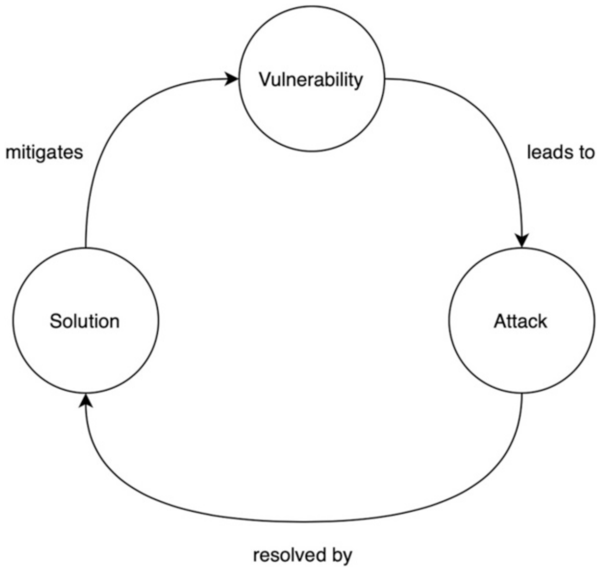


**Fig. 1.8** Generic vulnerabilities, attacks, and solutions cycle

to create taxonomies in similar contexts. Additionally, the taxonomy can aid in examining the security of PLSs from multiple perspectives. Secondly, academics can benefit from this taxonomy by identifying security trends and patterns in PLCs and using them to organize their research outcomes. Industry professionals can also use the taxonomy to target frequent PLC vulnerabilities, analyze existing solutions, and improve security measures. The taxonomy can also serve as a checklist to ensure a system is free of vulnerabilities mentioned in the taxonomy, indicating improved security. While this taxonomy already classifies vulnerabilities, attacks, and solutions in industrial PLC software, its significance will increase as researchers expand it to include other unknown categories.

In our taxonomy, the relation most frequently mentioned is the one between authentication vulnerabilities and the exploitation of these vulnerabilities by denial of service (DoS) attacks. A correct authentication mechanism is necessary for identifying legitimate users in the system. Without it, attackers can intrude into the PLC system and launch a DoS attack, where more traffic is sent to the system than it can handle.

The DoS attack also uses encryption vulnerabilities and a lack of anomaly detection. Encryption vulnerabilities are exploited because sensitive information, such as passwords, is sent in clear text instead of encrypted format, enabling the attacker to enter the PLC system. The lack of a detection mechanism allows attackers to use a DoS attack, and the system is unaware of it because it does not implement detection mechanisms to identify abnormal traffic and restrict it.

**Table 1.7** PLC security vulnerabilities, attacks, and solution mapping

| No | Vulnerabilities | Attacks | Challenge Response Mechanism | Deep Packet Inspection | Demilitarized Zone | Detection Mechanisms | Encryption | Firmware Verification Tool | Monitoring | Firewall | Prediction Model | Open PLC | SEABASS | User Authentication | VPN |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Authentication | Authentication Bypass | | | | | | | | | | * | * | * | |
| | | Dictionary | | | | * | | | | | | | | | |
| | | Phishing | | | | | | | | | | | | | |
| | | Interception | | | | | | | | | * | | | | * |
| 2 | Encryption | Interception | | | | | * | | | | | | * | | |
| 3 | Network | Denial of Service (DoS) | | * | | * | | | * | * | | | | | * |
| | | Stuxnet | | * | | * | | | * | * | | | | | * |
| | | Man in the Middle | | * | | * | | | * | * | | | | | * |
| | | Replay | | * | | * | | | * | * | | | | | * |
| | | SQL Injection | | * | | * | | | | * | | | | | * |
| | | Brute Force | | * | | * | | | | * | | | | | * |
| | | Havex | | * | * | * | | | | * | | | | | * |
| | | Phishing | | | | | | | | | | | | | * |
| | | Duqu | | | * | * | | | * | * | | | | | * |
| | | Maroochy | | | * | * | | | * | * | | | | | * |
| | | Interception | | | | * | | | | | * | | | | |

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | Operating System | PLC-PC worm | | | | * | | | | | | * | | | |
| | | Stealth command modification | | | | * | | | | | | * | | | |
| | | Data Execution | * | | | | | | | | | * | | | |
| 5 | Buffer Overflow | False Data Injection | * | | | * | | | | | | * | | | |
| 6 | Lack of Anomaly Detection | Denial of Service (DoS) | | | | * | | | * | * | | | | | * |
| | | Start/Stop | * | | | * | | | | | | | | | |
| 7 | Hardware and Firmware | False Data Injection | | | | | | * | | | | * | | | |
| | | Firmware Modification | | | | | | * | | | | * | | | |
| 8 | Input Validation | Stealth command modification | * | | | * | | | | | | | | | |
| | | Data Execution | * | | | | | | | | | | | | |
| | | Start/Stop | | | | * | | | | | | | | | |
| | | Control Logic Attack | * | | | | | | | | * | | | | |
| 9 | Human Issues | Data Execution | | | | * | | | | | * | | | | |
| 10 | Access Control | False Data Injection | | | | * | | | | | | | * | * | |
| | | Data Execution | * | | | | | | | | | | | * | |

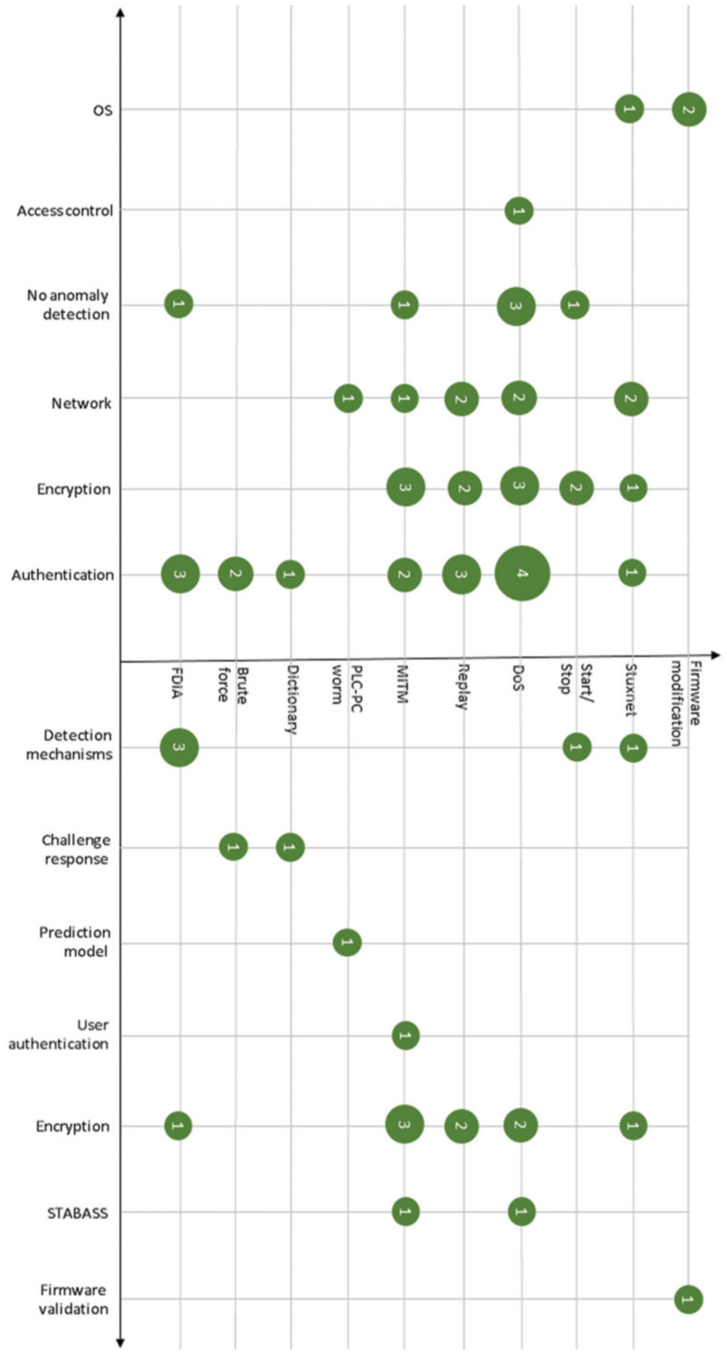


**Fig. 1.9** Relational display of vulnerabilities, attacks, and solutions

The man-in-the-middle and replay attacks also exploit encryption vulnerabilities. The attacks are executed by intercepting the communication between two devices and either listening or modifying the information being sent. Without encryption, attackers can easily read the data, although they cannot necessarily compromise the system. However, if the communication is encrypted, the data in the packet would require decryption by the user, making it much more difficult for attackers to access the PLC system.

The absence of an authentication mechanism makes it possible for false data injection attacks to exploit the authentication vulnerability. This is because it is difficult to determine if the exchanged messages are authentic or false without such a mechanism in place. Implementing cryptographic signatures can help authenticate messages, ensuring the data is valid. In addition to exploiting encryption vulnerabilities, the man-in-the-middle and replay attacks also exploit authentication vulnerabilities. Although challenge-response authentication mechanisms can mitigate encryption vulnerabilities, they are currently lacking. Attackers exploit this vulnerability to launch successful attacks. Another type of attack that exploits authentication vulnerabilities is the brute force attack. Since single-factor authentication is typically used, a brute-force attack can be successful by simply finding the password. The lack of a two-factor authentication mechanism can make such attacks successful.

The prevalent vulnerabilities identified in the context of FDIA are authentication and lack of anomaly detection. Correspondingly, detection mechanisms and encryption are the commonly suggested solutions. In contrast, the primary vulnerability in the case of MITM attack is the absence of encryption, and most solutions revolve around encryption mechanisms. The replay attack is vulnerable to several issues, including lack of authentication, lack of encryption, and network problems. The most popular countermeasures focus on encryption mechanisms.

## 1.7 Validity Threats

In this section, we discuss the potential validity threats that may arise in our study and the measures we have taken to mitigate them. According to Wohlin's categorization [27], validity threats are broadly classified into four types: construct validity, internal validity, external validity, and conclusion validity. We discuss each of these threats and the steps we have taken to minimize their impact on our study.

**Construct validity** Construct validity refers to the relationship between the data collected and the research questions. In our study, we mitigated this threat by defining a search string using the PICO criteria to ensure that the extracted data would answer our research questions. Additionally, we included all relevant keywords related to our study in the search string. The two libraries we selected are reputable sources in the field of PLC engineering, further increasing the construct validity of our study.

**Internal validity** To address internal validity, we established protocols for both the mapping study and the taxonomy, following established guidelines as a blueprint for conducting the study. This helps to control for external variables that may affect the outcomes of the study.

**External validity** To ensure external validity, we aimed to collect a comprehensive set of papers relevant to our study by using both automatic search and snowballing methods. This approach allowed us to include a wide range of publications that covered different aspects of PLC security, thus increasing the generalizability of our findings to other studies.

**Conclusion validity** To address the threat of conclusion validity, we took several measures. Firstly, we documented our study's process systematically and transparently, allowing others to repeat it and obtain the same results. Additionally, we aimed to minimize the potential impact of any new information that might be added during the time gap between our searches. Moreover, we defined a comprehensive data extraction form and classification framework to ensure that the study's results could be replicated.

## 1.8 Conclusions and Relation to DevOps

To get an overview of the existing research on security vulnerabilities, attacks, and security solutions in industrial PLC software, we performed a mapping study and developed a taxonomy for PLC-based security vulnerabilities, attacks, and solutions. The main goal of this work was to bring to the forefront the main vulnerabilities that malicious actors could exploit to gain access and attack the PLC system. This study can benefit academics and researchers who work with PLCs and focus on security. The taxonomy can help with an initial categorization of the most common vulnerabilities, attacks, and solutions.

Our results can be used by engineers working with security in DevOps. Using specific solutions and monitors identified using our taxonomy, one can use these prior to deployment in the design phase as predictors or oracles. For example, one can use specific detection mechanisms to perform test assessment and verification during development and to ensure security during the operational phase. This happens in DevOps when missing and vague security requirements identified by the monitors are added to the security requirements. The main advantage of the DevOps approach using our taxonomy is that it can be used further for tighter integration between design verification activities on executable PLC systems and runtime monitoring of such industrial systems.

## 1.9 Annex: Primary Studies

[S1] Jeong, E., Park, J., Oh, I., Kim, M., and Yim, K. (2020, July). Analysis on Account Hijacking and Remote Dos Vulnerability in the CODESYS-Based PLC Runtime. In International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (pp. 457–467). Springer, Cham.

[S2] Sarkar, E., Benkraouda, H., and Maniatakos, M. (2020, October). I came, I saw, I hacked: Automated Generation of Process-independent Attacks for Industrial Control Systems. In Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (pp. 744–758).

[S3] Gönen, S., Sayan, H. H., Yılmaz, E. N., Üstünsoy, F., and Karacayılmaz, G. (2020). False data injection attacks and the insider threat in smart systems. Computers and Security, 97, 101955.

[S4] Lee, J. C., Choi, H. P., Kim, J. H., Kim, J. W., Jung, D. U., Shin, J. H., and Seo, J. T. (2020). Identifying and Verifying Vulnerabilities through PLC Network Protocol and Memory Structure Analysis.

[S5] Khadpe, M., Binnar, P., and Kazi, F. (2020, July). Malware Injection in Operational Technology Networks. In 2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT) (pp. 1–6). IEEE.

[S6] Negi, R., Dutta, A., Handa, A., Ayyangar, U., and Shukla, S. K. (2020, June). Intrusion Detection and Prevention in Programmable Logic Controllers: A Model-driven Approach. In 2020 IEEE Conference on Industrial Cyberphysical Systems (ICPS) (Vol. 1, pp. 215–222). IEEE.

[S7] Bytes, A., and Zhou, J. (2020, October). Post-exploitation and Persistence Techniques Against Programmable Logic Controller. In International Conference on Applied Cryptography and Network Security (pp. 255–273). Springer, Cham.

[S8] Son, J., Noh, S., Choi, J., and Yoon, H. (2019). A practical challenge-response authentication mechanism for a Programmable Logic Controller control system with one-time password in nuclear power plants. Nuclear Engineering and Technology, 51(7), 1791–1798.

[S9] Lee, T., Kim, S., and Kim, K. (2019, October). A Research on the Vulnerabilities of PLC using Search Engine. In 2019 International Conference on Information and Communication Technology Convergence (ICTC) (pp. 184–188). IEEE.

[S10] Yao, Y., Sheng, C., Fu, Q., Liu, H., and Wang, D. (2019). A propagation model with defensive measures for PLC-PC worms in industrial networks. Applied Mathematical Modelling, 69, 696–713.

[S11] Gonzalez, D., Alhenaki, F., and Mirakhorli, M. (2019, March). Architectural security weaknesses in industrial control systems (ICS) an empirical study based on disclosed software vulnerabilities. In 2019 IEEE International Conference on Software Architecture (ICSA) (pp. 31–40). IEEE.

[S12] Yılmaz, E. N., Sayan, H. H., Üstünsoy, F., Gönen, S., and Karacayılmaz, G. (2019). Cyber security analysis of DoS and MitM attacks against PLCs used in smart grids.

[S13] Yoo, H., and Ahmed, I. (2019, June). Control logic injection attacks on industrial control systems. In IFIP International Conference on ICT Systems Security and Privacy Protection (pp. 33–48). Springer, Cham.

[S14] Pavesi, J., Villegas, T., Perepechko, A., Aguirre, E., and Galeazzi, L. (2019, November). Validation of ICS Vulnerability Related to TCP/IP Protocol Implementation in Allen-Bradley Compact Logix PLC Controller. In International Congress of Telematics and Computing (pp. 355–364). Springer, Cham.

[S15] Zhang, W., Jiao, Y., Wu, D., Srinivasa, S., De, A., Ghosh, S., and Liu, P. (2019). Armor PLC: A Platform for Cyber Security Threats Assessments for PLCs. Procedia Manufacturing, 39, 270–278.

[S16] Ghaleb, A., Zhioua, S., and Almulhem, A. (2018). On PLC network security. International Journal of Critical Infrastructure Protection, 22, 62–69.

[S17] Alves, T., Das, R., and Morris, T. (2018). Embedding encryption and machine learning intrusion prevention systems on programmable logic controllers. IEEE Embedded Systems Letters, 10(3), 99–102.

[S18] Alves, T., and Morris, T. (2018). OpenPLC: An IEC 61,131–3 compliant open source industrial controller for cyber security research. Computers and Security, 78, 364–379.

[S19] Lee, M., Choi, G., Park, J., and Cho, S. J. (2018, July). Study of Analyzing and Mitigating Vulnerabilities in uC/OS Real-Time Operating System. In 2018 Tenth International Conference on Ubiquitous and Future Networks (ICUFN) (pp. 834–836). IEEE.

[S20] Yılmaz, E. N., and Gönen, S. (2018). Attack detection/prevention system against cyber attack in industrial control systems. Computers and Security, 77, 94–105.

[S21] Ylmaz, E. N., Ciylan, B., Gönen, S., Sindiren, E., and Karacayılmaz, G. (2018, April). Cyber security in industrial control systems: Analysis of DoS attacks against PLCs and the insider effect. In 2018 6th International Istanbul Smart Grids and Cities Congress and Fair (ICSG) (pp. 81–85). IEEE.

[S22] Ng, J., Keoh, S. L., Tang, Z., and Ko, H. (2018, February). SEABASS: Symmetric-keychain encryption and authentication for building automation systems. In 2018 IEEE 4th World Forum on Internet of Things (WF-IoT) (pp. 219–224). IEEE.

[S23] Davidson, C. C., Andel, T., Yampolskiy, M., McDonald, J. T., Glisson, B., and Thomas, T. (2018). On SCADA PLC and Fieldbus Cyber-Security. In 13th International Conference on Cyber Warfare and Security (pp. 140–149).

[S24] Alves, T., Morris, T., and Yoo, S. M. (2017, December). Securing scada applications using openplc with end-to-end encryption. In Proceedings of the 3rd Annual Industrial Control System Security Workshop (pp. 1–6).

[S25] Abbasi, A., Holz, T., Zambon, E., and Etalle, S. (2017, December). ECFI: Asynchronous control flow integrity for programmable logic controllers. In Proceedings of the 33rd Annual Computer Security Applications Conference (pp. 437–448).

[S26] Pfrang, S., Meier, D., and Kautz, V. (2017, September). Towards a modular security testing framework for industrial automation and control systems: Isutest. In 2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA) (pp. 1–5). IEEE.

[S27] Wardak, H., Zhioua, S., and Almulhem, A. (2016, December). PLC access control: a security analysis. In 2016 World Congress on Industrial Control Systems Security (WCICSS) (pp. 1–6). IEEE.

[S28] Corbò, G., Foglietta, C., Palazzo, C., and Panzieri, S. (2016, October). Smart behavioural filter for SCADA network. In International Conference on Industrial Networks and Intelligent Systems (pp. 101–110). Springer, Cham.

[S29] Stone, S. J., Temple, M. A., and Baldwin, R. O. (2015). Detecting anomalous programmable logic controller behavior using rf-based hilbert transform features and a correlation-based verification process. International Journal of Critical Infrastructure Protection, 9, 41–51.

[S30] Sandaruwan, G. P. H., Ranaweera, P. S., and Oleshchuk, V. A. (2013, December). PLC security and critical infrastructure protection. In 2013 IEEE 8th International Conference on Industrial and Information Systems (pp. 81–85). IEEE.

[S31] Clark, A., Zhu, Q., Poovendran, R., and Başar, T. (2013, June). An impact-aware defense against Stuxnet. In 2013 American Control Conference (pp. 4140–4147). IEEE.

[S32] Milinković, S. A., and Lazić, L. R. (2012, November). Industrial PLC security issues. In 2012 20th Telecommunications Forum (TELFOR) (pp. 1536–1539). IEEE.

[S33] Masood, R., and Anwar, Z. (2011, December). SWAM: Stuxnet worm analysis in metasploit. In 2011 Frontiers of Information Technology (pp. 142–147). IEEE.

[S34] Olmstead, S., Stites, J., and Aderholdt, F. (2011, October). A layer cyber security defense strategy for smart grid programmable logic controllers. In Proceedings of the Seventh Annual Workshop on Cyber Security and Information Intelligence Research (pp. 1–1).

[S35] Valentine, S., and Farkas, C. (2011, August). Software security: Application-level vulnerabilities in SCADA systems. In 2011 IEEE International Conference on Information Reuse and Integration (pp. 498–499). IEEE.

[S36] Serhane, A., Raad, M., Raad, R., and Susilo, W. (2018, August). PLC code-level vulnerabilities. In 2018 International Conference on Computer and Applications (ICCA) (pp. 348–352). IEEE.

[S37] Yang, W., and Zhao, Q. (2014, August). Cyber security issues of critical components for industrial control system. In Proceedings of 2014 IEEE Chinese Guidance, Navigation and Control Conference (pp. 2698–2703). IEEE.

[S38] McMinn, L., and Butts, J. (2012, March). A firmware verification tool for programmable logic controllers. In International Conference on Critical Infrastructure Protection (pp. 59–69). Springer, Berlin, Heidelberg.

[S39] Basnight, Z., Butts, J., Lopez Jr, J., and Dube, T. (2013). Firmware modification attacks on programmable logic controllers. International Journal of Critical Infrastructure Protection, 6(2), 76–84.

**Acknowledgments** This work has received funding from H2020 under grant agreement No. 737494, from Vinnova through the SmartDelta project and from KKS through the ACICS project. This work was partially supported by the Austrian Science Fund (FWF): I 4701-N.

## References

1. C.C. Davidson, T. Andel, M. Yampolskiy, J.T. McDonald, B. Glisson, T. Thomas, in *13th International Conference on Cyber Warfare and Security* (2018), pp. 140–149
2. G. Sandaruwan, P. Ranaweera, V.A. Oleshchuk, in *2013 IEEE 8th International Conference on Industrial and Information Systems* (IEEE, 2013), pp. 81–85
3. S.E. Valentine Jr., PLC code vulnerabilities through SCADA systems, Doctoral dissertation (2013). Retrieved from https://scholarcommons.sc.edu/etd/803
4. W.C. Yew, PLC device security – tailoring needs, GIAC (GSEC) Gold Certification (2019). https://www.giac.org/research-papers/37612/
5. J. Weiss, *Protecting Industrial Control Systems from Electronic Threats* (Momentum Press, 2010)
6. J. Falco, J. Falco, A. Wavering, F. Proctor, *IT Security for Industrial Control Systems* (Citeseer, 2002)
7. K. Stouffer, J. Falco, K. Scarfone, NIST Spec. Publ. **800**(82), 16 (2011)
8. Guide to Industrial Control Systems (ICS) Security. Standard, National Institute of Standards and Technology (2015)
9. X. Pan, Z. Wang, Y. Sun, J. Cybersecur. **2**(2), 69 (2020)
10. H. Wu, Y. Geng, K. Liu, W. Liu, in *IOP Conference Series: Materials Science and Engineering*, vol. 569 (IOP Publishing, 2019), p. 042031
11. I. Pekaric, C. Sauerwein, S. Haselwanter, M. Felderer, Comput. Stand. Interfaces **78**, 103539 (2021)
12. R.C. Nickerson, U. Varshney, J. Muntermann, Eur. J. Inf. Syst. **22**(3), 336 (2013)
13. M. Usman, R. Britto, J. Börstler, E. Mendes, Inf. Softw. Technol. **85**, 43 (2017)
14. P. Bourque, R. Dupuis, A. Abran, J.W. Moore, L. Tripp, IEEE Softw. **16**(6), 35 (1999)
15. K. Petersen, R. Feldt, S. Mujtaba, M. Mattsson, in *12th International Conference on Evaluation and Assessment in Software Engineering (EASE) 12* (2008), pp. 1–10
16. E.N. Yılmaz, S. Gönen, Comput. Secur. **77**, 94 (2018)
17. H. Wardak, S. Zhioua, A. Almulhem, in *2016 World Congress on Industrial Control Systems Security (WCICSS)* (IEEE, 2016), pp. 1–6
18. J. Pavesi, T. Villegas, A. Perepechko, E. Aguirre, L. Galeazzi, in *International Congress of Telematics and Computing* (Springer, 2019), pp. 355–364
19. T. Alves, T. Morris, Comput. Secur. **78**, 364 (2018)
20. E.N. Yrlmaz, H.H. Sayan, F. Üstünsoy, S. Gönen, G. Karacayilmaz, Cyber security analysis of DoS and MitM attacks against PLCs used in smart grids, in *2019 7th International Istanbul Smart Grids and Cities Congress and Fair (ICSG)*, Istanbul, Turkey (2019), pp. 36–40
21. A. Clark, Q. Zhu, R. Poovendran, T. Başar, in *2013 American Control Conference* (IEEE, 2013), pp. 4140–4147

22. R. Masood, Z. Anwar et al., in *2011 Frontiers of Information Technology* (IEEE, 2011), pp. 142–147
23. M. Khadpe, P. Binnar, F. Kazi, in *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)* (IEEE, 2020), pp. 1–6
24. H. Yoo, I. Ahmed, in *IFIP International Conference on ICT Systems Security and Privacy Protection* (Springer, 2019), pp. 33–48
25. W. Zhang, Y. Jiao, D. Wu, S. Srinivasa, A. De, S. Ghosh, P. Liu, Procedia Manuf. **39**, 270 (2019)
26. S.A. Milinković, L.R. Lazić, in *2012 20th Telecommunications Forum (TELFOR)* (IEEE, 2012), pp. 1536–1539
27. C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, *Experimentation in software engineering* (Springer Science & Business Media, 2012)

# Chapter 2
# Natural Language Processing with Machine Learning for Security Requirements Analysis: Practical Approaches

Andrey Sadovykh, Kirill Yakovlev, Alexandr Naumchev, and Vladimir Ivanov

**Abstract** Analyzing security requirements is a tedious task. Quite often they are spread around requirements specifications or specified in a very generic form. The experts have to make sure to extract all the security requirements and properly detail by applying the best practices from appropriate standards such as OWASP ASVS, STIG, or IEC62443. The requirements are specified in various forms, most commonly as statements in natural language. Natural language processing (NLP) has been applied for many years in requirements engineering (RE) for many analysis tasks. However, until recently, the performance on NLP methods on the RE tasks has been questionable. In this chapter, we outline the state of the art in the NLP methods in RE and in particular analysis of security requirements as well as provide practical recipes application of modern transfer learning architectures to several important RE tasks illustrated with an example.

**Keywords** Security requirements · Requirements engineering · Natural language processing · Machine learning · Dataset · Classification · Semantic search · VeriDevOps

## 2.1 Introduction

Requirements engineering (RE) is a crucial element in the software development to meet customers' expectations for a software product that should be delivered on time and within a budget. Practically, RE enables to capture users' needs for the system to be developed by transferring these needs into precise and clear statements that will

A. Sadovykh (✉) · K. Yakovlev · A. Naumchev · V. Ivanov
SOFTEAM, Ivry-sur-Seine, France
e-mail: andrey.sadovykh@softeam.fr; kya@softeam-rd.eu; anau@softeam-rd.eu; viv@softeam-rd.eu

A. Sadovykh et al. (eds.), *CyberSecurity in a DevOps Environment*,
https://doi.org/10.1007/978-3-031-42212-6_2

be the basis for design, development, and validation [2]. Requirements engineering helps to define the right scope of a project and address all nonfunctional properties such as security starting from early stages of the design and implementation. This is often reported as an important approach to improve productivity, speed up the delivery time, and decrease the costs of software development. In the case of cybersecurity, building-in the right security mechanisms, addressing the potential vulnerabilities, and following the standardized guidelines is the most common way to protect the critical assets by the company, its customers, and system users. Nowadays, when the security threats are discovered on a daily basis, analyzing and ensuring implementation of security requirements have become of ultimate importance. It is reported that the security mechanisms have to be built into the system starting from the architecture stages, since retrofitting these important aspects into the system is extremely expensive.

The requirements engineering includes many activities. One of them is a requirements analysis involving the requirements categorization among other activities. Identifying and placing a requirement to the right category, for example, security, may help to address important concerns by the right specialists as early as possible in the project life cycle. The current approach for automating quality control in a continuous manner with the Continuous Integration and Continuous Delivery (CICD) pipelines with DevOps practices has brought many benefits with respect to security properties verification. However, the challenge of "left-shifting" the security verification to early stages of development and even to the requirements analysis still remains due to the lack of automation.

One of the challenges in creating the automation for requirements analysis and verification is that the prevailing method to specify requirements is natural language. Although the formalization approaches exist and help in validating the requirements, in practice, the requirements statements styles and lexical structures vary a lot. Natural language processing (NLP) proposes a number of methods to deal with texts and receive the information, analyze semantic similarity, etc. NPL has been applied for many years to requirements engineering offering many practical benefits though the performance of those legacy solutions is questionable. With the appearance of deep neural networks and transformer architectures in 2018, NLP made a huge leap forward in terms of performance. Many researchers have started to apply those methods to requirements engineering and obtained interesting results.

In this chapter, we outline the legacy and novel NLP methods as applied to requirements engineering. We illustrate the application of transformers architecture with our own experiments and prototypes for several RE tasks. Finally, discuss the applicability of these methods and potential usage in the continuous cybersecurity assessment in the DevSecOps context.

## 2.2 Security Requirements Engineering

An important part of developing any system is ensuring a required level of security. Security needs are usually associated with some resources or assets involved in a system that stakeholders naturally want to protect from any harm. In particular, assets are considered as all the information resources that are stored or accessed by the system or physical resources such as computers. In some cases, assets may consist of other assets, e.g., system backups are a good example. Despite that empirical evidence is not fully convincing, it appears that appropriate security requirements would have as positive an impact on system security as sufficient general requirements would have on system development success.

In order to integrate security within requirement engineering, we usually have to consider separately security requirements [3]. Special research showed that early analysis of security requirements can be beneficial in the context of software development as this may enable cost reductions in the area of 12–21% [4]. Usually security requirements are processed as functional requirements that can considerably influence system architecture. In practice, this requires a specific security expertise. Essentially, security requirements have to be specially processed independently from other requirements. However, the whole process of manual identification or extraction of security-related requirements from an entire requirement specification is very complex and error-prone, causing the need for automatic analysis. This is associated with several practical challenges. Firstly, there is no exact definition for security requirements, since different people may interpret security requirements in various ways. In practice, different industry subjects – organizations – define security requirements based on their own conventions and templates. Secondly, the intrinsic ambiguity of natural language makes it even more complicated to identify security requirements. Primarily, various people may use different syntax and terms to define or describe security requirements [5, 6].

The main point is that security cannot be considered as just a quality requirement, as it is difficult to answer whether a problem is security-related or not. Usually stakeholders do not tolerate any kind of risks. The main task of security requirement engineering is to identify and document requirements for developing secure software systems. The identification of security requirements heavily depends upon the context of system and analyst's assumptions. These assumptions can be explicit or implicit and relate to expectations over system or environment behavior with a significant impact on the security of a system. Considering the framework of security, its goals, and assumptions, one can define security requirements as constraints on the functions of the system, where these constraints operationalize one or more security goals. In other words, security requirements engage security goals by constraining the system's functional requirements. Security requirements, like functional requirements, are prescriptive, providing a specification to achieve the desired effect [3].

Based on these goals, we conclude that we want to prevent any threat or potential attack aimed at our assets. We consider *assets* as something that is

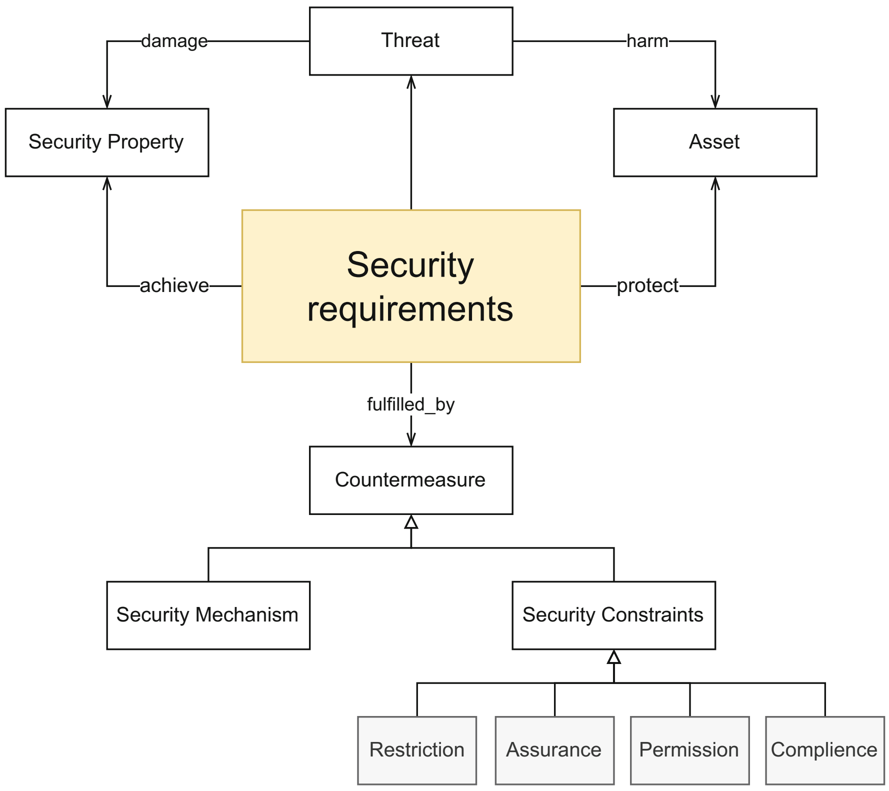


**Fig. 2.1** A conceptual model for security requirements [5]

valuable to an organization (e.g., resources, data) and typically is a main concern of security requirements. Practically, a *security property* determines a security characteristic (e.g., confidentiality, availability) that indicates a security objective that a requirement intends to achieve. A *threat* is an undesired event that a swindler may potentially exploit to attack the system, harms assets or its respective security properties. In this context, a *countermeasure* is considered as a protective measure prescribed to meet the *security requirements*. The *countermeasure* may be represented by a *security mechanism* as well as a set of *security constraints*. Each of the above concepts (Fig. 2.1) contributes with a specific perspective of the *security requirement* definition [5].

In order to follow this structure and achieve initial goals, we must somehow determine whether those requirements have been satisfied. This is difficult for quality requirements in general, while security requirements present additional challenges [3]. An important element of the requirements engineering is associated with the role of natural language (NL). Despite that there is no proof that natural language is the best option, multiple evidences show that it is the most common way of expressing requirements in the industry practice. The dominance in describing and specifying software and system requirements in natural language was also confirmed by the recent research [7]. Therefore, based on the past and current empirical evidence, we can safely assume that NL will continue to serve as the common way of expression for requirements in the future as well [8]. Conceptually that implies that solutions should deal with problems like lexical, syntactic, and pragmatic problems that natural language poses for requirements engineering. It is stated that the biggest problem is ambiguous semantics, which remains a common challenge for practitioners arguing that the source of trouble is the information from which the requirements must be formulated [9]. Consequently, computer-aided software engineering for processing natural language looks promising in the context of requirements analysis [8].

## 2.3 Natural Language Processing for Requirements Engineering (NLP4RE)

Applying NLP techniques, which are very well suited for comprehensive linguistic analysis, seems natural in the context of the engineering approach that suggests using linguistic tools to narrate descriptions of user requirements. NLP is a field that addresses various approaches in which computers can deal with natural, that is human, language. Usually, NLP deals with techniques for analyzing, representing naturally occurring texts for the purpose of achieving human-like language processing for range of tasks or applications [10, 11]. This has led to the emergence of a separate field, i.e., NLP4RE of applying NLP to support requirements engineering process as well as various tasks at different RE phases [12]. Dealing with the inputs to the RE process is a complicated task, as it requires to analyze a wide variety of documents. Such documents might include different artifacts like interview transcripts, codes of practice, standards, legislation, etc. In practice, the methods for RE automation greatly differ depending on the stage of RE they are applied at. To illustrate, at later stages, such as requirements validation, the methods deal mainly with documents that are products of the RE process, whereas at early stages the methods typically process raw information [9]. By applying those methods, the engineers intend to solve different kinds of tasks like detecting language issues, identifying key domain concepts and establishing traceability links among requirements, etc. However, when we split the developed NLP solutions by problem that they solve, they are mainly focused on *detection*, *classification*, *clustering*, *patterns extraction*, and *modeling* [12]. Those instruments are intended to increase analysts' productivity when working with requirements.

Let us outline the key method categories. **Detection** typically deals with ambiguities in requirements to make them clearer and unequivocal. The range of problems may include detection of different lexical issues from the debatable usage of grammatical rules, to the occurrence of vague phrases (e.g., after some time), weak verbs (e.g., may, might), and the appearance of syntactic ambiguities. In addition, some specific tasks such as following to predefined templates and recognizing equivalent requirements can also be included in this task, as the main goal is still to maintain a correctness to requirements texts. **Classification** task in ML is usually associated with predicting a categorical class [13]. As for the context of RE, this task aims at classifying different categories of requirements. For example, we can classify requirements based on their functional category or based on their quality category, to identify nonfunctional requirements that may be hidden within functional ones. Another example is applying classification to users' feedback in order to identify new requirements referring to specific features of interest possibly including a sentiment analysis. **Extraction** generally tries to retrieve some specific single or multi-word terms from requirement texts for domain or project glossaries, as requirements usually contain complex terms that are not commonly used. Those extracted glossaries may be further applied for other problems including consistency checking, classification, modeling, or product comparison. **Clustering** or cluster analysis, as its name suggests, is focused on organizing data, in our case, documents or a set of textual requirements into some cohesive subsets or clusters. This method focuses on organizing the data into meaningful and useful information. **Modeling** relates to the extraction task but with some additional usage of extracted data like generation of unified modeling language (UML) models to support analysis, design, feature synthesis in product-line engineering, generation of models for early requirements and generation of software tests to maintain a necessary security level [12].

In addition to the abovementioned generic problems that NLP solves in requirements engineering, one can outline several approaches that are entirely focused on the security context. Despite the lack of studies in this area, we can highlight an initial progress in developing and implementing such systems. Security risks can be analyzed through different perspectives that will define a practical context of the problem. **Vulnerability detection** is focused on identifying vulnerable code sequences by analyzing software code prior to deployment. The approach concentrates on applying NLP techniques to code to prevent or identify various vulnerabilities in the code. **Vulnerability repair** tries to transform a vulnerable code into a non-vulnerable code by learning from a set of source examples. Millions of lines of legacy code are analyzed to identify the ways to improve security. When a new class of vulnerability is found, the training dataset for patches and fixes is quickly updated. This is intended for creating an automated system that can clean code with certain types of vulnerabilities that would allow to treat efficiently large software repositories. Finally, **specification analysis** assumes that we can deal with security risks in product before the code is even written. Recent advances in NLP have provided experts methods to automatically process vulnerability descriptions or product specifications to assess security risks. Instead of code we can apply methods to documents and text vulnerabilities in this paradigm to ensure a required security level for the developed software [14]. Our main interest is associated with this security perspective.

The subsections below outline specific NLP approaches for classification, extraction, as well as advanced machine learning architectures, e.g., transfer learning as applied to RE tasks and security requirements analysis.

### *2.3.1 Statistical and Classical Machine Learning Methods*

NLP addresses several practical problems in the area of requirements engineering. To start, let us consider the problem of distinguishing functional requirements from nonfunctional ones. Abad et al. [15] propose text preprocessing as the main tool of dealing with that task. To address the generalization problem for the input requirements texts, they proposed to preprocess the texts and replaced all context-based names related to products and users with general keywords, such as "PRODUCT" and "USER," respectively. Then they apply the Part-Of-Speech (POS) tagger of the Stanford Parser [16] to assign parts of speech to each word in each requirement. In the next step, they extract some trivial features including number of adjectives, number of adverbs and number of cardinals, as well as specific metrics, such as number of degree adjectives to adverbs. In addition, for each feature they define its rank based on the probability of its occurrence in the requirements. The final feature list for the processed dataset consists of the following nine features: number of cardinals, adverbs, adjectives, modal words, determiners, verbs, prepositions, singular nouns, and plural nouns. In [15] the authors compare results of six different algorithms and use a simple decision classifier to achieve an extra 4.5% accuracy of classifying functional and nonfunctional requirements. This effect becomes even more visible for classifying groups of requirements. Abad et al. insist that Binarized Naive Bayes works best for classifying nonfunctional requirements.

Another example of NLP application to requirements engineering is identifying critical features in specifications. Boutkova et al.[17] propose a lexical analysis based technique that could help automate the identification of features in specifications. They propose to extract features in a semi-supervised fashion by applying certain Part-of-Speech (POS) tagging approaches. The whole process is divided into several steps. At the first step, the user chooses the specification in which the features must be found. At the second step, requirements from the chosen specification get decomposed into individual words, and only nouns are left; this step requires lemmatization of each word. At the final step, the user should evaluate candidates list and choose features for the feature model. The main problem is that the experiments were conducted for German – a morphologically complex language. This approach generates a lot of false positives that need further analysis.

It is possible to improve the performance by combining different NLP developments from different disciplines. Malhotra et al.[18] proposed an approach combining NLP, ML, and graph analysis. This approach identifies appropriate narrative structures that may underlie the security requirements of industry standards and publicly available software documents. First, the authors of [18] apply text processing that includes tokenization, sentence splitting, POS tagging, morphological analysis, and noun phrase chunking. Then they create an ontology to define connections between words, phrases, and concepts. They construct features from key narrative structures – phrases, such as "user must register," "user must contain a password," "password must have complexity" using a special tool called

Protégé [19]. Subsequently, each of these processing structures is used to determine the relationships among features such as “encryption” or “authentication.” After, it is checked whether the requirement sentence is found among gold standard requirements. That way it is determined whether an organization follows those standards.

The idea of checking whether security requirements conform to specific standards was also presented in Hayrapetian et al. [20]. This study might be considered as an advancement of the previous study, focusing on empirically evaluating conformance of security requirements to specific standards such as ISO and OWASP. The main goal was to assess completeness and ambiguity by creating a bridge between the requirement documents and its compliance to standards. For this purpose, they proposed a unique two-stage architecture. Initially every statement within a standard is evaluated against every statement within a test document. To maintain robustness of an entailment assessment, they proposed nine different configurations and digested each pair through those components. Each configuration consisted of the Linguistic Analysis Pipeline and Entailment Decision Algorithms from Excitement Open Platform [21]. The entailment decision and confidence results from each transaction were collected along with other data about the transaction, such as the statements involved, entailment configuration used, processing type (e.g., parallel), and the time duration of the comparison. These annotations were used as features during the neural network model training phase to design a classifier to further determine whether the entailment results for a statement pair indicate a “complete,” “ambiguous,” or “none” match, with respect to the corresponding semantic meaning. This approach allowed to achieve 0.79 in terms of F1-score.

One of the main challenges on the way of making all-purpose NLP methods is a problem of generalization of a model to be applied to several domains. Li et al. [5] presents the idea of creating a model that could generalize security requirements extraction for all domains. They stated that the main source of good detection lays in a good theoretical basis and tried to construct ontology specifically for security requirements. They defined a set of linguistic rules and security keywords that are normally used to describe security requirements and used them to train classifiers applying classical ML algorithms. They proposed a specific approach that involves a two-level preprocessing with a conceptual layer and linguistic layer. The process of matching the linguistic features consists of three steps: generate parse trees, keyword matching, and linguistic rule matching. Each step is explained in detail as a part of text processing to a feature vector. They decided to compare different algorithms like decision tree (DT), Naive Bayes classifier (NBC) and logistic regression (LR) using six different datasets. Results showed that precision/recall differs among datasets. Only DT and LR showed promising characteristics. In particular, the average F1-score of all classifiers trained with DT was approximately 0.77. For the case of classifying security requirements from different domains, when training data was used from one document set and the test data from the other, this approach showed 0.75 in precision and 0.58 in recall. The authors argued that their approach behaves significantly better than the existing approach and potentially can give promising results. They also argued that the main challenge was that different people, including security experts, can have various diverse definitions of security requirements.

Another example of dealing with the generalization problem is the work by Wang et al. [22]. They address aspects of generalization from a different perspective such as creating all-domain classifiers. The authors developed methods for extracting security requirements for open-source projects (OSS). They stated that previously proposed approaches were unsuitable for this kind of projects due to their specifics. Notably, requirement specifications in OSS projects are usually organized by functionality, with nonfunctional (NFR) requirements scattered widely across multiple documents. Hereby there is no exact boundary to distinguish between FRs and NFRs. Moreover, the requirements stored in issue tracking systems are unstructured and seldom obey grammar and punctuation rules. The authors proposed to rely not on the text but on different external resources. To define features, they applied a stack of several sources that then were used as an input for a linear classifier based on linear discriminant analysis (LDA). Initially each requirement is processed by information processing component (IPC) to obtain so-called metrics. Metrics are information about a requirement extracted by IPC, which includes complexity and external resources. Complexity is extracted from comments of the project assuming that higher intensity of discussion might be associated with vulnerabilities. In its turn, external resources are the links and other references provided by stakeholders where they discuss rationale for refinements and explain their solutions. Subsequently, this information is digested directly by four regression models: Comment Complexity Regression Model (CRM), Stakeholder Complexity Regression Model (SRM), Security URLs Regression Model (URM), and Security Commits Regression Model (CiRM). In addition, the authors apply NFR classifier (NFR-C) and CVE ID Detector (CID). Each regression model generates a weight between 0 and 1 for each requirement that signify the likelihood whether this requirement is a security requirement. In order to summarize weights from NFR-C, CID, and all RMs, the authors applied a linear discriminant function in a binary setting that indicate whether a requirement is security one or not. They were able to achieve F1-scores of 0.83, 0.88, and 0.81 for Axis2, Drools, and GeoServer projects, respectively, which looks promising given the relative simplicity of the proposed approach.

In 2017, RE Data Challenge event was conducted in relation to the problem of requirements extraction and classification. This event produced a set of NLP4RE studies. Kurtanovic et al. [23] used the dataset from the challenge [24] to solve the problem of binary classification for functional (FR) and nonfunctional (NFR) requirements. Simply, they transformed a multiclass dataset into a binary case. Unlike previous papers, authors did make a research of an effect from applying only word features and automatically chosen features form binary and multiclass classification. The whole approach is based in the support vector machines. As a result, they achieved an F1-score of 0.92 for binary case classifying FR and NFR. As for classifying security requirements in a binary case, the effect was a bit worse. If applying only words features, F1-score was about 0.88 and 0.74 with applying all kinds of features. They found that POS tags are among the most informative features, with cardinal number being the best single feature. As an additional aspect, authors argued that only word features provide higher recall for classifying NFRs than employing additional syntax and meta-data features but lower precision accordingly.

Pérez-Verdejo et al. [25] explored applicability of several machine learning algorithms for classification of software requirements and issue reports. The classifier that reached the highest weighted geometric mean was TPOT (Tree-Based Pipeline Optimization Tool), with 0.8363, followed by the RandomForestClassifier classifier with 0.82. Pérez-Verdejo et al. report, however, that these models showed significant difficulties in classifying issue reports (often expressed in the form of informal text), as compared to human experts. It was difficult for automated classifiers to obtain results greater than 0.3 on classifying requirements-related issues.

Mir Khatian et al. [26] focus on the prediction of the requirements classification of NFRs (nonfunctional requirements) by using supervised machine learning (ML) algorithms followed by comparative analysis on five different ML algorithms: decision tree, k-nearest neighbor (KNN), random forest classifier (RFC), and Naïve Bayes and logistic regression (LR). The exhaustive results of the comparative analysis conducted by Mir Khatian et al. demonstrate that the performance of the LR algorithm is the best of all algorithms with high prediction rates and 75% accuracy. The Naïve Bayes resulted in 66% accuracy, the decision tree provided 60% accuracy, the RFC provided 53% accuracy, and KNN – 50% accuracy. According to the study of Mir Khatian et al., the LR algorithm should be preferred for the prediction of the classification of NFRs.

### *2.3.2 Deep Learning*

Deep learning and transfer learning are the most recent areas in NLP. Deep learning is assumed as a sub area of neural networks in ML and is popular for vision-based classification and NLP tasks. Deep learning is based on the representation-learning methods obtained by applying nonlinear modules that transform a representation at one level into a higher, more abstract level [27]. Zhang and Wallace [28] proposed convolutional neural network [29] for the purpose of sentence classification. They provided a simple method that is based on Word2Vec representations [30] of each word with applying set of consecutive convolution filters. Specifically, the process starts with tokenized sentence which is converted to a sentence matrix, the rows of which are word representations. By this approach authors achieved significant accuracy improvements comparing with a baseline on all datasets. Similar approach was applied to the context of requirements classification. Winkler et al. [31] applied the same principle to the DOORS requirements database. Specifically, they applied it for the binary classification task to differentiate requirement from information sentences. This approach was able to classify requirements with a precision of 0.73 and a recall of 0.89 and information with a precision of 0.90 and a recall of 0.75 accordingly. The authors argued that performance could be improved by increasing the amount of training data as well as by improving the quality of requirement specifications. A similar approach was applied to the previously mentioned NFR dataset. Dekhtyar et al. [32] presented an idea of combining two methods that are very popular at the moment, Word2Vec, and convolutional neural networks (CNN). They used two datasets, SecReq dataset [6] and the quality attributes (NFR) dataset [24], to compare results of applying Word2Vec with CNN with a baseline approach. The goal was to observe the performance of CNNs on these datasets compared to the baselines and measure the impact of pretrained Word2Vec embeddings on the model. As a baseline method, they considered already mentioned approach based on Naive Bayes classifier [6] with TF-IDF and word counts as feature vectors. For SecReq dataset applying Word2Vec provided an overall boost in scores. By applying 30 filters with 100 training epochs, they scored an F1-score of 91.34%. This configuration allowed to achieve an overall improvement up to 13.5% compared to the baseline. For NFR dataset Word2Vec again contributed comparable improvement with 50 filters and 100 epochs accordingly. The authors stated that CNN classifiers can be successfully applied on relatively small collections of requirement documents to identify various requirements properties.

### *2.3.3 Transfer Learning*

Recently the transfer learning method was applied as a new promising approach to deal with generalization problem. Hey et al. [33] stated that the performance of existing automatic classification methods decreases when applied to unseen projects, because requirements usually vary in formulation and style. This means that such systems are impractical to use, as they are either overfit for a specific dataset, which is heavily relying on wording and sentence structure or require a processing step (usually manual) for new text samples. Moreover, usually, authors do not report whether their approaches are able to generalize or do not generalize sufficiently to be practically applicable. One reason is the lack of available training data in the requirements engineering community. The authors stated that possible solution can be found in a transfer learning. Nowadays transfer learning approaches are heavily used in NLP. They are trained on huge datasets to capture underlying concepts and meanings of natural language texts. Afterward they can be adapted and fine-tuned to a specific task. Authors stated that this helps to overcome the problem of generalization, as these approaches promise both better performance and generalizability with less training data. That is achieved by fine-tuning of Bidirectional Encoder Representations from Transformers (BERT) [34], a language model based on deep learning. BERT, which is pretrained on a large text corpus, can be fine-tuned for specific tasks by providing only a small amount of input data such as requirements classification in our case. For experiments NFR dataset [24] was chosen as a gold standard coming from RE Data Challenge'17. This whole process is common for BERT-based studies. Specifically, BERT model is applied with a single layer of NN for classification purposes. The resulting model is called NoRBERT. The authors provided a detailed information about experiments, which would help to replicate their results in future. For binary tasks, NoRBERT achieved comparable results with an F1-score of 90% for functional and 93% for nonfunctional requirements. As it was expected, BERT-based method outperformed all existing approaches at the moment. Specifically, NoRBERT outperforms all approaches that do not preprocess the data, and, at the same time, the problem with unweighted data does not significantly impact performance. Thus, the transfer learning approach clearly increases the performance for classifying requirements. As for security requirements, NoRBERT was able to achieve about 0.91 in F1-score given multilabel classification, which might look promising for a further application.

As Ajagbe et al. [35] point out, BERT underperforms on domain-specific tasks. They introduce BERT4RE, a BERT-based model retrained on requirements texts, aiming to support a wide range of requirements engineering (RE) tasks, including classifying requirements, detecting language issues, identifying key domain concepts, and establishing requirements traceability links. Ajagbe et al. also fine-tune BERT4RE for the task of identifying key domain concepts and conclude that BERT4RE achieves better results than the BERT base model for the same task. Ameri et al. [36] and Ranade et al. [37] take further steps in BERT fine-tuning. Ameri et al. fine-tune BERT using a corpus of labeled sequences from industrial control systems device documentation collected across a range of vendors and devices. They claim improvement in classification accuracy from 76% to 94.4% accuracy as compared with the original BERT architecture. Ranade et al. [37] fine-tune BERT on a cybersecurity corpus from open-source unstructured and semi-unstructured cyber threat intelligence (CTI) data, using masked language modeling (MLM) to recognize specialized cybersecurity entities. They evaluate the resulting model using downstream tasks that can benefit security operations centers (SOCs). Ranade et al. claim the fine-tuned model outperforms the base BERT model in the domain-specific MLM evaluation.

Li et al. [38] apply BERT to treat the problem of poor generalization of other requirements classification models. They use apply graph attention network (GAT) to mine the syntactical structure of requirements and take it into account in their model. Li et al. evaluate the resulting approach, DBGAT, on the PROMISE datasets. They report up to 91% F1-score for the classification task on already seen projects, and up to 88% F1-score – for unseen-before projects.

## 2.4 Practical Examples of NLP4RE

As part of demonstration, we have created several prototypes with different functionality focused on solving various NLP tasks in requirements engineering for analysis of security requirements. This section follows the path NLP process for security requirements analysis starting from an unstructured document to extraction of requirements, identification of security requirements and finally semantic search of relevant security countermeasures.

### *2.4.1 ReqExp: Requirements Extraction from a Text*

In this section we discuss the first NLP task to address the extraction problem. We analyzed a well-known requirements dataset, PURE [39], manually extracted requirements and non-requirements sentences, and trained a prototype, ReqExp, based on the [34] architecture. We discuss the overall approach and our results in the following part.

The requirements extraction problem may be addressed by the NLP classification methods. One can notice that text classification is applied in many NLP applications, such as spam filtering, email categorization, information retrieval, web search, document classification, etc. Usually, it means assigning predefined categories to a textual sequence [40]. It has to be noted that requirements are commonly specified in the form of a sentence rather than in a form of unstructured phrases. We can thus translate the extraction problem into the classification of a given sentence into the class of possible "requirements" sentences. In the context of classification of requirements statements, it is needed to consider the whole sentence since the context of the requirement is critical for analysis. Despite the existence of several standardized lexical forms for requirements specification, requirement sentences oftentimes do not follow these standards formally. This imposes a necessity to have a system that could extract requirements in any lexical structure form.

For requirement engineering, it is important to process entire documents and extract the requirements sentences with a high precision. In NLP terms it relates to two subsequent steps: (1) extracting all sentences from a document and cleaning them and (2) classifying sentences to the "requirements" and "non-requirements" classes. The classification task approach and experimental results are addressed in our previous research [41]. The classification is based on the state-of-art model architectures, i.e., Bidirectional Encoder Representations from Transformer (BERT). However, from the perspective of requirements engineering, the requirements extraction from documents has several peculiarities that we outline below.

As it usually stands, any classification or extraction process starts with a preprocessing stage where the system processes an input text and produces a set of objects ready for an NLP analytical task, i.e., sentence classification in our case. It usually involves removing stop words, typographical symbols, punctuation marks or even correcting lexical mistakes, etc. This preprocessing may be manually conducted before experiments, e.g., dataset preparation. In a production environment, in an application, this preprocessing is automated each time before applying models. Technically, this automation is defined by analyzing the manual preprocessing experiments. For example, the stopwords and punctuation are removed during experiment and training of the model, but also this is done automatically inside the final application in the production environment.

Nowadays, research teams usually deal with pretrained models (like BERT) that completely shift their focus from a model architecture design to a thorough and well-justified fine-tuning of a chosen model. Practically, we adapt model parameters to a certain domain as well as to a task itself, i.e., the classification in our case.

However, this does not imply significant change in the model architecture. The transformers approaches such as BERT allow shifting the NLP research more to the steps of theoretical justification of the method choice and dataset preparation. The first step, a method justification, is usually based on domain expertise, previous experiments as related to the task that we want to solve. We chose transformers model architectures and BERT as they outperform the other approaches on the complex NLP tasks.

The dataset preparation is the most time-consuming, sophisticated, but at the same time the most important stage, especially considering the complexity of the RE area. The dataset preparation is executed by running several activities problems simultaneously:

1. Collecting an appropriate set of samples (e.g., sentences) that most accurately describe our domain.
2. Defining domain classes with clear lexical and semantic differences.
3. Selecting samples in each class that contain all the necessary features for each class.

From the first sight, it appears as an overwhelming task, but it can be decomposed in a set of smaller steps. The process itself starts with researching potential sources to construct the required dataset. Unfortunately, the RE sphere has not been popular among NLP researchers so far that resulted in some shortage of nicely designed datasets for classification or extraction tasks. In our case we focused on designing a dataset, which contains sentences that can be clearly binary classified in requirement/non-requirement classes. We observed several critical issues related the following questions:

- How to exactly identify what a requirement class should look like knowing that writing styles and lexical structures in different areas greatly differ?
- What should a non-requirement class contain? Should that be just random sentences or something else?

We have analyzed a significant set of research papers that outlines earlier in this chapter and in [41] to define what sources are suitable for a dataset construction. Ferrari et al. [39] (Table 2.1) proposed a comprehensive corpus of documents that contains publicly available documents from different projects and software engineering areas. Overall, it contains 79 documents with a focus on applying for designing NLP systems.

Authors argue that this dataset fairly fits for various tasks such as requirements categorization, ambiguity detection, equivalent requirements identification, etc. Moreover, this corpus includes documents with different peculiarities as well as a lexicon with the widespread writing style of requirements. However, the corpus needs to be analyzed and processed specifically in order to create a dataset suitable for requirements sentence classification.

We applied PURE corpus as our main source of both requirements and non-requirements sentences for designing our dataset. Initially documents were presented in the form of raw text in different formats, structures, and writing styles. We

**Table 2.1** Comparing PURE corpus with Brown corpus [39]

| Indicator | PURE | Brown |
|---|---|---|
| Number of tokens | 865,551 | 1,034,378 |
| Number of lexical words | 522,444 | 542,924 |
| Vocabulary size (lexical words) | 21,791 | 46,018 |
| Vocabulary size (stems) | 16,011 | 29,846 |
| Number of sentences | 34,268 | 57,340 |
| Average sentence length (tokens) | 25 | 18 |
| Average sentence length (lexical words) | 15 | 10 |
| Lexical diversity | 0.031 | 0.054 |

extracted 7745 requirement/non-requirement sentences, where 4145 were requirements and 3600 were non-requirements from 30 documents. Requirement sentences were extracted from the appropriate sections of the documents. Usually, each document provides some structural elements like table of contents and requirement-focused sections with appropriate titles or contextual footnotes. The structural elements are usually numbered by unique ids, sometimes explained by the authors. In other cases, it could be useful to rely on lexical elements that are usually inherent for requirements such as modal verbs, e.g., "must," "should," "could," etc. In essence, we applied a process of identifying requirements and non-requirements that had been described in a previous study by Abualhaija [42].

Making experiments with models usually starts with dividing the datasets in subsets in order to train, test, and validate a model. For this purpose, we separated the obtained sentences into train, test, and validation sets by following 70%, 20%, and 10% proportions accordingly. To bring the experiment closer to the application domain and preserve consistency, it was decided to select an entire group of sentences from every document only to one specific subset. Thus, some of the documents were applied for the model training and different ones for validation. Additionally, we conducted a specific analysis to identify a set of documents that could fulfill abovementioned proportion constraints (70%, 20%, 10%).

Training process, which is the fine-tuning in our case, usually happens in a straight way. As we already mentioned, our focus was not training a new model rather adapting parameters to our domain. In the case of classification, we do not touch the middle layers that were trained initially by model architecture's authors. However, we still needed to adapt this pretrained model to our classification task, since transformers do not directly provide classification but are designed to generate numerical representation of digested sentences and words only. For that purpose we augmented our architecture with additional neural network layers of an appropriate size.

Figure 2.2 depicts a conceptual design for the requirements extraction from a document by applying a transformers model architecture to requirements sentences classification. Usually text samples, e.g., words, sentences, or blocks of text, exist in the form of a string, which is preprocessed by a group of methods for

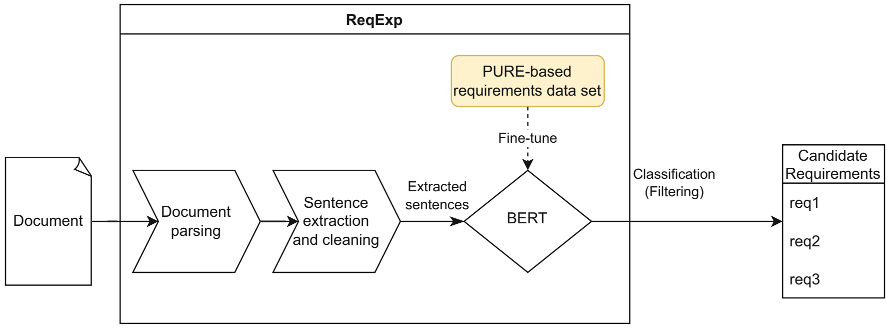


**Fig. 2.2** Requirement classification in ReqExp prototype

**Table 2.2** ReqExp experiment results

| Model | F1 | Precision | Recall |
|---|---|---|---|
| Fasttext | 0.81 | 0.72 | 0.93 |
| ELMO | 0.83 | 0.78 | 0.88 |
| BERT | 0.86 | 0.92 | 0.80 |

cleaning. Afterward this sentence is digested by an architecture-specific tokenizer that encodes our string into a model input, specifically transforming a text into some smaller pieces like words or tokens. These tokens are sequentially evaluated on each layer of the model architecture. The result is a probability for a sentence to belong to the requirements class. One should decide on an appropriate threshold to assign the input of the requirements sentence to the "requirement" or "non-requirement" class, for example, a 0.5 or any other optimal value by the AUC-ROC analysis (Table 2.2).

The experiment results have to be assessed using appropriate metrics. We select three major classification metrics: precision, recall, and F1-score. Those metrics are perceived as the golden standard in many research areas of statistical and mathematical methods including machine learning and deep learning. Let us explain what those metrics do specifically mean in the requirements engineering domain:

1. Precision indicates the confidence in detecting requirements, since it compares the number of requirements rightly guessed (TP) with the number of requirements wrongly guessed (FP), i.e., Precision = TP / (TP + FP). The better the precision is, the greater is the confidence in the requirements guessed rightly.
2. Recall is a proxy to assess the number of undetected requirements, since it compares the number of detected requirements (TP) with the number of undetected requirements (false negatives or FN), i.e., Recall = TP / (TP + FN).
3. Finally, a combination of the precision and recall, F1-score, is calculated as follows F1 = 2*Precision*Recall / (Precision + Recall). In other words, F1-score conveys the balance between the precision and the recall and may serve as a good measure of overall performance of a specific model.

In our research we collected all those metrics to have a full picture of what each model is capable of as well as to provide a broad conclusion which model is better specifically in the context of requirements engineering. Initially, we expected that transformers would be leaders; however, it was important to explore the difference compared to the baseline models such classic Fasttext and ELMO. Our experiments showed that transformers have the highest potential for dealing with such uncommon contexts as software requirements. Specifically, the more advanced BERT model showed better results in almost all aspects, especially in terms of F1-score. The BERT model showed high **precision 0.92** but lower **recall 0.8**. Still precision and recall metrics behaved differently for Fasttext and ELMO models. Fasttext classifier showed impressive **recall 0.93** compared with other candidates. Presumably such property might be useful in cases when it is more important to extract most of the required sentences regardless of the larger number of false positives [41].

We recommend BERT-based architectures as a basis for classification (or extraction) tasks even without retraining the whole architecture and staying with a basic fine-tuning. However, BERT-based solutions are relatively resource demanding [43]. This must be considered carefully for real-world applications in the industry context. Our study showed that traditional less-demanding models may demonstrate acceptable results in the more constrained environments.

### *2.4.2 SeqReq: Security Requirements Classification*

In the context of the VeriDevOps project [1], our team participated in a study to design a system that could correctly identify security requirements in various texts or software specification documents. We manually classified a large dataset of security and non-security-related requirements and trained a specific prototype, SecReq, based on the DestilBERT [44] architecture. We present the overall approach and our results in the section below.

This task is again mainly a classification task, but the context is more specific since the classifier can only be applied within the restricted set of sentences, e.g., identifying security-related context in the set of requirements. Compared to the binary classification approach that we discussed earlier, the scope of the task is very narrow and requires a specific solution. Initially our goal was to design a system based only on machine learning methods to assess its capabilities for solving this task in an industrial context. Thereby, we focused only on solutions based on ML architectures like deep neural networks.

Several solutions exist for this NLP task; the selection has to be justified based on the analysis of the domain with regards to lexical and semantic properties as well as datasets. We conducted a preliminary analysis resulting in the following conclusions:

- There exist several definitions of security requirements, which however are not specific enough with respect to the lexical properties. Moreover, the statement styles vary a lot from company to company, industry to industry, author to author.

- A specific dataset is needed for classifying security requirements. In our cases this dataset had to be constructed from several sources.
- As with general requirements, there is a clear shortage of well-designed datasets presumably due to its narrow perspective.

Our analysis showed that this task can again be reduced to the binary classification task as in the previous case for extracting the requirements sentences. However, a specific dataset needed to be constructed that would contain security-related requirements.

We analyzed the available datasets and security text corpora. As a result, we collected several datasets and other security-related text sources from different business domains:

1. The first one is the **PROMISE dataset**, which is considered as a benchmark dataset widely used in literature. This dataset originated from the RE'17 Data Challenge. It contains 625 sentences among which 375 are nonfunctional requirements of 12 classes (like availability, reliability, scalability, security, etc.) and 255 are functional requirements [45].
2. Additionally, we found requirements in official documentation. **CCHIT** [46] published criteria for several products developed in 2006 and 2007. These criteria consist of 283 requirements, including security-related statements, which are also useful for our task (Certification Commission for Healthcare Information Technology Work Groups, 2007).
3. **SRS Concordia** corpus was constructed as a reconsideration of the PROMISE dataset. Authors indicated several problems of the PROMISE dataset with respect to artifact types and sentences, which may have multiple or no labels. They proposed their own corpus, which contains 6 documents that can be transformed into 3064 manually labeled sentences [47].
4. **OWASP** Application Security Verification Standard [48] provides a collection of security requirements for web applications. The number of security requirements differs from one version of the standard to another. In order to gather most of the statements present in the standard, we created a dataset as a union of OWASP ASVS v3.0.1 and v4.0. This dataset contains 496 security-relevant sentences.

Nevertheless, it was unclear how well each dataset covered the "security-related" class of requirements. We decided to consolidate all the extracted samples and aggregate them in a separate dataset. That way, we obtained 2328 text samples, where 804 sentences represented security requirements, while the other 1518 sentences were non-security requirements. Additionally, we augmented this dataset with 651 security requirement sentences that we collected by manually labeling previously mentioned PURE-based dataset. As a validation step, we checked both datasets for possible intersections and removed them to avoid any possible bias in a further assessment.

Specifically for the stage of experiments, we applied the PURE set as our train dataset, whereas the combined set was used for testing. We conducted experiments with various model architectures to find the most efficient one. Initially, we focused on already well-established transformer architectures like BERT [34], MPNET [49], and their variations. Both transformer architectures, BERT and MPNET, are pretrained on a huge corpus of English-language texts, and this facilitates building higher-capacity models for a wide variety of tasks.

Nowadays, one can notice that the industry has made a leap forward by privileging libraries of pretrained deep learning models as, for example, HuggingFace [50] that includes SBERT [51]. These libraries propose a multitude of solutions for almost any well-known task in NLP, including classification, text generation, text similarity, etc. SBERT is dedicated to the analysis of complete sentences. In addition, those libraries provide many pretrained and ready-to-use architectures by vendors from different application areas, for example, healthcare, finance, and information technologies. These models may represent a complete solution for a variety of NLP tasks or be considered as a starting point for creating a specific adaptation. To design a solution for the security requirements extraction task, we considered pretrained architectures that address NLP tasks like in our context. After several experiments we identified that among other architectures the most promising model for our goal was a special version of DistilBERT [44] that was fine-tuned on a special Stanford Sentiment Treebank [52] dataset designed for a sentiment classification task.

To address the narrow context of the security requirement extraction problem, we decided to apply a combination of models that are executed sequentially in the following pipeline (Fig. 2.3):

- **Stage 1 "ReqExp"**: Fine-tuned BERT for identifying requirements in the text as described in the previous section.
- **Stage 2 "SecReq"**: DistilBERT for identifying security requirements in a set of requirements that was specifically tuned for this task.

The overall structure of the pipeline is shown in the diagram below. Figure 2.3 depicts all the main stages with datasets used for fine-tuning, as well as inputs and outputs for each stage. Stage 1 "ReqExp" starts with uploading a document where some preprocessing steps happen to extract all the possible text from a document

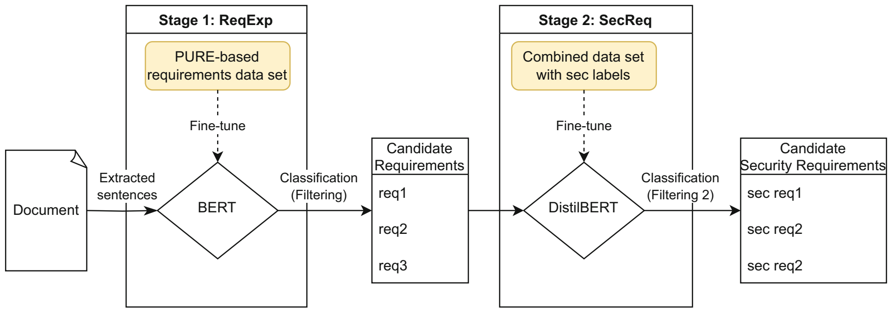

Stage 1: ReqExp
PURE-based requirements data set
Fine-tune
Document
Extracted sentences
BERT
Classification (Filtering)
Candidate Requirements
req1
req2
req3
Stage 2: SecReq
Combined data set with sec labels
Fine-tune
DistilBERT
Classification (Filtering 2)
Candidate Security Requirements
sec req1
sec req2
sec req2

**Fig. 2.3** Security requirements classification

and transform it to a text with specific lexical elements that are sentences in our case. Afterward, one-by-one every sentence is digested to the first stage to make a filtering and extracting requirements statements from the input text samples. At the second stage, the obtained list of requirements is processed by the fine-tuned DistilBERT to distinguish security requirements from other types of requirements. During the process of fine-tuning of the SecReq stage, we were able to achieve an **F1-score of 0.86** using this version of the model.

Let us consider an illustrative example by NIST from [53]. While examining this source document, one may notice that the authors specified the single security requirement in the related section: *The software system defined in this SRS must follow industry recommended practices for secure software development. At a minimum, the software development must practice the principle of least privilege for defining access-level requirements of the software system and its associated services. The production-release version of the software system must pass an automated dynamic application security testing tool (e.g., HP WEBINSPECT).* In contrast, our prototype [54] extracted the following security-related requirements sentences:

1. *The system must have at least a super-user role and a user role defined for accessing and interacting with the system.*
2. *Distributing manufacturing data across an enterprise requires the curation and management of the data within a repository and end-user services to access the data.*
3. *The UI for administrative tasks must include the ability to manage users, manage groups, manage permissions, manage data templates, manage group assignments, and manage query templates.*
4. *SuperUserRole/001 maintains all VDS and QDR back-end system configurations, SuperUserRole/002 maintains all VDS and QDR schemas and templates, SuperUserRole/003 maintains all user groups and user accounts, and SuperUserRole/004 Maintains all QDR predefined queries*
5. *The user must be presented with a login page when accessing any page, or expect a landing or front page, as an anonymous user.*

We consider this result as relevant and important, since the prototype indicated several security sensitive functional requirements that have to be analyzed by dedicated specialists mastering the security approaches for development and validation.

### *2.4.3 STIGSearch: Semantic Search for Security Technology Implementation Guides*

In the example in the previous section, one may also notice that the original requirements are extremely generic. At most, this specification suggests to follow "best practices." The document leaves a great flexibility to developers to select a standard architecture and apply related recommendations. This approach, quite common in industrial practice, represents a great vulnerability since it can easily lead to omission of numerous concrete guidelines that were created by analyzing the vulnerabilities. One of the approaches is to de-generalize the security requirements and map them to right practical recommendations. For this purpose, we created a specific prototype, STIGSearch, based on the SBERT [51] architecture for semantic search of relevant countermeasures in the Security Technology Implementation Guides (STIGs) [55] database. We present the overall approach and our results in the section below.

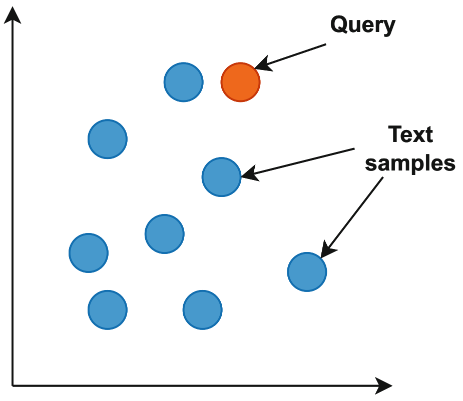
Often the security requirements are expressed as a need to comply with a specific standard, such as Security Technology Implementation Guides (STIGs) [55], Web Application Security Project (OWASP) recommendations (OWASP Application Security Verification Standard, n.d.) or standards with an extremely narrow perspective like ISAIEC 62443 [56]. One of the analysis goals is to locate the relevant standard requirements, guidelines, or recommendations that relate to the security requirements that are specified by an engineer specifically for a project. This may help to relate the project requirements and corresponding implemented features with security standards. In the NLP context, the semantic search task is dedicated to identifying semantic proximity among lexical entities. This approach may be beneficial as a solution for the requirements-to-standard mapping problem.

Traditionally, search techniques are designed based on word computation models and, in some cases, enhanced by the link analysis. In contrast, the semantic search technique extends the information retrieval with entity and knowledge retrieval, instead of looking into the keyword matching frequency only. In other words the semantic search addresses the search task from a different perspective by assessing the meaning of words that are formalized and represented in machine processable format [57].

Practically semantic search is focused on improving the search experience by understanding the content of the search query. For example, in contrast to traditional search engines which only find content based on lexical matches, semantic search can also find synonyms. The idea behind semantic search is to transform all entries in a corpus, whether they be sentences, paragraphs, or documents, into a vector space. In our case we deal with high-dimensional representations – tokens or embeddings that we discussed in the previous sections. At the search time, a query is embedded into the same vector space in order to find the closest embeddings from the source corpus. The core idea is that these closest entries should have a high semantic similarity with the query (Fig. 2.4) [58].

In order to compare the vector representation of a query with all the elements in the source corpus, one shall use a special metric, which is called a cosine similarity. Cosine similarity measures the proximity between two vectors of an inner product space. It is measured by the cosine of the angle between two vectors and determines to which degree two vectors are pointing in the same direction [59]. This property is quite useful in various tasks when comparing text pairs, e.g., sentence with sentence, word with word, etc. Figure 2.5 illustrates the cosine similarity with a two-dimensional example.

**Fig. 2.4** Semantic search principle



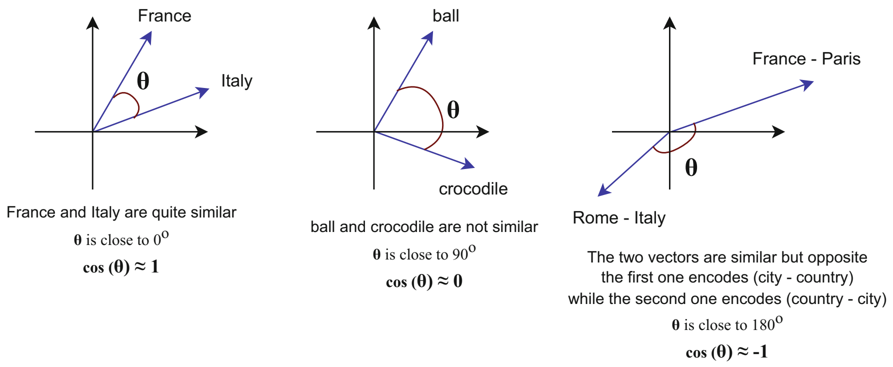

France
Italy
θ
France and Italy are quite similar
θ is close to 0°
cos (θ) ≈ 1
ball
crocodile
θ
ball and crocodile are not similar
θ is close to 90°
cos (θ) ≈ 0
France - Paris
Rome - Italy
θ
The two vectors are similar but opposite
the first one encodes (city - country)
while the second one encodes (country - city)
θ is close to 180°
cos (θ) ≈ -1

**Fig. 2.5** Cosine similarity [60]

In our approach, we proposed to design a search solution for a combined database that included security requirements from STIGs and IEC 62443 standards.

The core challenge in the designing process was mainly to identify what representations could be most adequate to make the search method locate the most relevant set of standard requirements, given a complex context that we are dealing with in this case.

We have analyzed various options that include such advanced models as MPNet and DistilROBERTA that were pretrained specifically for semantic search [61]. As it was already mentioned, for this NLP task, these are the vector representations that really matter. Therefore, searching process technically becomes straight following (Fig. 2.6):

1. Transform the source corpus (e.g., STIGs) using the chosen model architecture and save it in an optimal format, e.g., CSV file.
2. Transform the query using the same model as the model chosen to represent the source corpus.

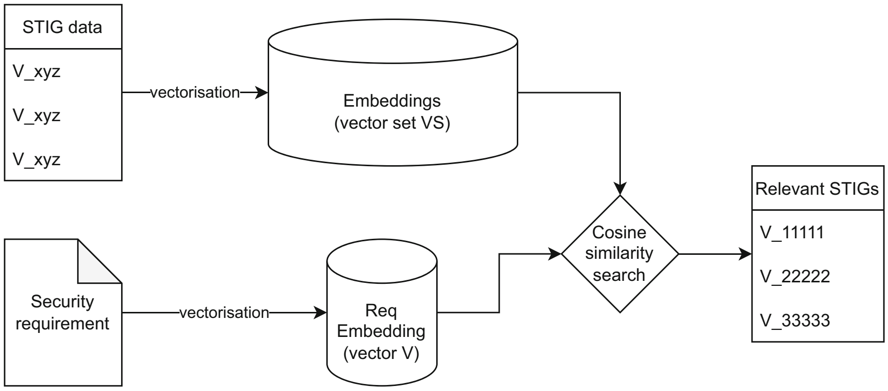


**Fig. 2.6** Semantic search method applied to security requirements

3. Apply cosine similarity to the query vector representation and each element in the vector space representing the source corpus.
4. Finally, sort all the obtained metrics and extract indexes of most similar elements in the source corpus.

In order to evaluate the search method performance and applicability in the industry context, we designed a separate prototype including several pretrained models, which implement equivalent functionality – semantic search in security standards dataset. The principle remains the same regardless of the model architecture; however, the query results will differ, and their quality has to be assessed manually by the users.

Let us consider again the illustrative example that we provided in the previous section, the requirements specification by NIST [53] states: the software development must practice the principle of least privilege for defining access-level requirements of the software system and its associated services. This requirement statement is quite generic and that may lead to a misinterpretation. If we apply the semantic search in IEC 62443 database [56], we obtain, e.g., the following:

1. *Components shall provide an authorization enforcement mechanism for all users based on their assigned responsibilities and least privilege.*
2. *Components shall provide, or integrate into a system that provides, the capability to enforce password minimum and maximum lifetime restrictions for all users.*
3. *Components shall provide an authorization enforcement mechanism for all identified and authenticated users based on their assigned responsibilities.*
4. *Components shall provide the capability to limit the use of resources by security functions to protect against resource exhaustion.*
5. *Components shall provide the capability to perform or support integrity checks on software, configuration, and other information as well as the recording and reporting of the results of these checks or be integrated into a system that can perform or support integrity checks.*

This shows that the original requirement by NIST can be semantically mapped to several standard requirements. Furthermore, if we apply the semantic search in the STIG database for Microsoft Windows platform, we may locate a number of recommendations, e.g.:

1. *User rights assignments must meet minimum requirements.*
2. *The operating system must employ a deny-all, permit-by-exception policy to allow the execution of authorized software programs.*
3. *Permissions for program file directories must conform to minimum requirements.*
4. *The roles and features required by the system must be documented.*
5. *Unauthorized users are granted right to Act as part of the operating system.*
6. *Security configuration tools or equivalent processes must be used to configure and maintain platforms for security compliance.*
7. *Users with administrative privilege must be documented and have separate accounts for administrative duties and normal operational tasks.*

Each guideline provides concrete steps to check and fix a number of related issues on the operating system level. Thus, the software developer and the DevOps specialists may obtain the instructions for security the deployment of the system and notice a number of design patterns to be implemented in the software.

## 2.5 Discussion

In this paper we have presented several solutions for classical tasks in NLP4RE like requirements extraction and semantic search. Our focus was to design systems based on pretrained state-of-art model architectures.

The first problem was to design an extraction tool for requirements from documentation or text corpora related to the software specifications. This problem can be translated into a NLP classification task with an additional preprocessing step. Usually requirements exist in the form of sentences in a specification document. That meant that we needed to first extract and clean the sentences from a document, which was an additional tedious task by itself. Moreover, for classification with the NLP methods, it is required to have a clear definition of characteristics of "requirement" sentences that we need to extract. As it was stated earlier, the requirement statements vary a lot in form. This imposes a potential bias in a further assessment by the NLP models.

Classifiers based on BERT architecture played a central role in our solution. There exist different variants and enhancements for this architecture. For our experiments we applied several basic models. For the requirements extraction problem, we compared the BERT-based solution with other advanced methods like Fasttext and ELMO. Nevertheless, BERT clearly overperformed all those methods. To train this model, we prepared a special binary dataset that was based on PURE corpus and constructed by manually extracting sentence by sentence and then labeling it in accordance with its class. Although one can recognize a requirement sentence with a certain degree of confidence, the NLP approach also required non-requirements class sentences, which cannot be defined that easily. It should be noted that this separation on requirement and non-requirement classes can vary depending upon the theoretical basis authors take and hypothesis authors are relying on. Potential changes in hypothesis and in model architecture may significantly influence final results.

As an advancement of a classification problem, we also designed a classifier for identifying security requirements in software specification related texts. In this case we applied a pipeline consisting of two models – our previously trained model for requirements classification and the model for recognizing security-related requirements. The first model serves to extract a set of requirements statements. The second model filters out the security-related requirements with an additional binary classification step. For this case we created a combined dataset that is specifically labeled to distinguish security-related requirements from non-related ones. Relabeling PURE corpus-based dataset specifically for security context was not enough since the obtained dataset was rather limited. That required us to find more additional sources of security-based text samples for training our system. The goal was to achieve completeness and impartiality in our experiments. Due to inability to assess the class coverage, it might be seen as a biased approach. Still as with the previous problem, there is still no consistent and well-established guide to follow for labeling the security-related requirements.

Finally, in this paper we presented our solution for a searching engine to query requirements databases of the security guidelines and standards. Our method hypothesis was based on the semantic search theory, which today prevails in tasks of finding similar text elements, e.g., documents, sentences, etc. In this case, we followed a path by relying on high-dimensional representations of requirements sentences. We also applied model architectures adapted to a semantic search. The essence of the method is to apply a pretrained model and create a set of embeddings for the source dataset, i.e., STIG guidelines, IEC62443 requirements. The query sentence is vectorized for the same model architecture. The query embeddings and dataset vector space are compared using the cosine similarity method. The query/standard sentence pair with the highest cosine ranking is selected as the most relevant and similar text. We designed several prototypes with different pretrained model architectures in order to conduct a thorough evaluation in our future research.

We have identified several ways with the intent to improve our prototypes. As the baseline, the work on cleaning, augmenting, and validating the datasets has to continue. There is a need for an improved guide for labeling "requirements" statements and distinguishing "security-related" requirements. In addition, the labeling work by human specialists has to be cross validated, so that there is a high degree of coherence. In addition, the "security" class of requirements has to be further augmented so that it could be possible to filter individual categories of security requirements such as data integrity or authentication. Our preliminary studies of the current security requirements datasets have shown that many categories are underrepresented. Further on, the baseline ML model architecture can be retrained on the corpus of security-related text as it was illustrated in [36]. This may potentially improve the classification results on several NLP tasks. There is a difficulty in evaluating performance of the semantic search methods, since there is a lack of an objective quality metric – the semantic relevance is subjective. In this context, there is a need to integrate users' feedback to the evaluation and improvement of our methods.

## 2.6 Conclusions

In this chapter we have outlined the current state-of-the-art in NLP for requirements engineering and have given several practical examples of application of model transfer learning architectures for several security requirements analysis tasks. This area is important and is considered an entry point in the VeriDevOps project [1]. Indeed, the VeriDevOps methodology starts from automatically assessing the newcoming security requirements, e.g., vulnerability reports, users' requests, attacks, and anomalies detected. These requirements have to be properly categorized and mapped to the corresponding practices for specific design patterns or countermeasures implementation. Our work contributes to analysis of security requirements specified in natural language. First, we propose to evaluate an incoming request, e.g., document or a textual statement as related to security. Second, we offer a semantic search mechanism with the goal to map security requirements to an appropriate practice.

In this chapter we have applied the BERT architecture for requirements extraction and security requirements filtering as well as MPNet architecture with SBERT for semantic search over the cosine similarity metrics. The resulting prototypes ReqExp, SecReq, and STIGSearch integrated into the ARQAN tool set[54] demonstrate promising results though further evaluation and improvement may be required.

For providing better support on mapping requirements to recommendations and countermeasures, we explore possibilities to link user requirements with STIG recommendations and concrete implementations for the security tests. In particular, our research on the requirements verification automation with RQCODE [62] is an enabler that links the security requirements statements in natural language and security verification mechanisms such as tests.

In the requirements engineering domain, we experiment with the integration of our prototypes with various requirements modeling and management tools such as Modelio [63] and GitHub/GitLab issue trackers. In addition, the integration with the CI/CD tools over the specific pipeline mechanism helps to address the requirements analysis automation challenge that has been identified for the DevSecOps area [64].

**Acknowledgments** This work is partially supported by the VeriDevOps [1] project funded by the Horizon 2020 program under the grant agreement No. 957212 (VeriDevOps project).

## References

1. A. Sadovykh, G. Widforss, D. Truscan, E.P. Enoiu, W. Mallouli, R. Iglesias, A. Bagnto, O. Hendel, in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)* (2021), pp. 1330–1333. https://doi.org/10.23919/DATE51398.2021.9474185. ISSN: 1558-1101
2. P. Loucopoulos, V. Karakostas, *System Requirements Engineering* (McGraw-Hill, 1995)
3. C. Haley, R. Laney, J. Moffett, B. Nuseibeh, IEEE Trans. Softw. Eng. **34**(1), 133 (2008)
4. Hoo, K. Soo. Tangible ROI through secure software engineering. Secur. Bus. Q. (2001). https://cir.nii.ac.jp/crid/1571698600432996480
5. T. Li, Z. Chen, J. Syst. Softw. **165**, 110566 (2020)
6. E. Knauss, S. Houmb, K. Schneider, S. Islam, J. Jürjens, in *International Working Conference on Requirements Engineering: Foundation for Software Quality* (Springer, 2011), pp. 4–18
7. M. Kassab, C. Neill, P. Laplante, Innov. Syst. Softw. Eng.: A NASA J. (2014). https://doi.org/10.1007/s11334-014-0232-4
8. L. Mich, M. Franch, P.L. Novi Inverardi, Requir. Eng. **9**, 40 (2004). https://doi.org/10.1007/s00766-003-0179-8
9. P. Sawyer, P. Rayson, K. Cosh, IEEE Trans. Softw. Eng. **31**, 969 (2005). https://doi.org/10.1109/TSE.2005.129
10. D. Jurafsky, C. Manning, Instructor **212**(998), 3482 (2012)
11. E.D. Liddy, Natural language processing, in *Encyclopedia of Library and Information Science*, 2nd edn. (Marcel Decker, Inc., NY, 2001)
12. L. Zhao, W. Alhoshan, A. Ferrari, K. Letsholo, M. Ajagbe, E.V. Chioasca, R. Batista-Navarro, *Natural language processing for requirements engineering: a systematic mapping study*. ACM Comput. Surv. **54**(3), (2022)
13. T. Hastie, R. Tibshirani, J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* (Springer Science & Business Media, 2009)
14. S. Kommrusch, arXiv preprint arXiv:1912.06796 (2019)
15. Z.S.H. Abad, O. Karras, P. Ghazi, M. Glinz, G. Ruhe, K. Schneider, in *2017 IEEE 25th International Requirements Engineering Conference (RE)* (IEEE, 2017), pp. 496–501
16. C.D. Manning, M. Surdeanu, J. Bauer, J.R. Finkel, S. Bethard, D. McClosky, in *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations* (2014), pp. 55–60
17. E. Boutkova, F. Houdek, in *2011 IEEE 19th International Requirements Engineering Conference* (IEEE, 2011), pp. 313–318
18. R. Malhotra, A. Chug, A. Hayrapetian, R. Raje, in *2016 International Conference on Innovation and Challenges in Cyber Security (ICICCS-INBUSH)* (IEEE, 2016), pp. 26–30
19. N.F. Noy, M. Crubézy, R.W. Fergerson, H. Knublauch, S.W. Tu, J. Vendetti, M.A. Musen, in *AMIA... Annual Symposium Proceedings. AMIA Symposium* (2003), pp. 953–953
20. A. Hayrapetian, R. Raje, in *Proceedings of the 11th Innovations in Software Engineering Conference* (2018), pp. 1–11
21. B. Magnini, R. Zanoli, I. Dagan, K. Eichler, G. Neumann, T.G. Noh, S. Pado, A. Stern, O. Levy, in *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations* (2014), pp. 43–48
22. W. Wang, K.R. Mahakala, A. Gupta, N. Hussein, Y. Wang, J. Ind. Inf. Integr. **14**, 34 (2019)
23. Z. Kurtanović, W. Maalej, in *2017 IEEE 25th International Requirements Engineering Conference (RE)* (IEEE, 2017), pp. 490–495
24. J. Cleland-Huang, S. Mazrouee, H. Liguo, D. Port. NFR (2007). https://doi.org/10.5281/zenodo.268542
25. J.M. Pérez-Verdejo, Á.J. Sánchez-García, J.O. Ocharán-Hernández, E. Mezura-Montes, K. Cortés-Verdín, Program. Comput. Softw. **47**(8), 704 (2021)
26. V. Mir Khatian, Q. Ali Arain, M. Alenezi, M. Owais Raza, F. Shaikh, I. Farah, in *2021 1st International Conference on Artificial Intelligence and Data Analytics (CAIDA)* (IEEE, Riyadh, Saudi Arabia, 2021), pp. 7–12

27. Y. LeCun, Y. Bengio, G. Hinton, Nature **521**(7553), 436 (2015)
28. Y. Zhang, B. Wallace, arXiv preprint arXiv:1510.03820 (2015)
29. Y. LeCun, B. Boser, J.S. Denker, D. Henderson, R.E. Howard, W. Hubbard, L.D. Jackel, Neural Comput. **1**(4), 541 (1989)
30. T. Mikolov, K. Chen, G. Corrado, J. Dean, arXiv preprint arXiv:1301.3781 (2013)
31. J. Winkler, A. Vogelsang, in *2016 IEEE 24th International Requirements Engineering Conference Workshops (REW)* (IEEE, 2016), pp. 39–45
32. A. Dekhtyar, V. Fong, in *2017 IEEE 25th International Requirements Engineering Conference (RE)* (2017), pp. 484–489. https://doi.org/10.1109/RE.2017.26
33. T. Hey, J. Keim, A. Koziolek, W.F. Tichy, in *2020 IEEE 28th International Requirements Engineering Conference (RE)* (IEEE, 2020), pp. 169–179
34. J. Devlin, M.W. Chang, K. Lee, K. Toutanova, arXiv preprint arXiv:1810.04805 (2018)
35. M. Ajagbe, L. Zhao, in *2022 IEEE 30th International Requirements Engineering Conference (RE)* (2022), pp. 309–315
36. K. Ameri, M. Hempel, H. Sharif, J. Lopez Jr., K. Perumalla, J. Cybersecur. Privacy **1**(4), 615 (2021). https://doi.org/10.3390/jcp1040031. https://www.mdpi.com/2624-800X/1/4/31
37. P. Ranade, A. Piplai, A. Joshi, T. Finin, in *2021 IEEE International Conference on Big Data (Big Data)* (2021), pp. 3334–3342
38. G. Li, C. Zheng, M. Li, H. Wang, IEEE Access **10**, 30080 (2022)
39. A. Ferrari, G.O. Spagnolo, S. Gnesi, in *2017 IEEE 25th International Requirements Engineering Conference (RE)* (2017), pp. 502–505. https://doi.org/10.1109/RE.2017.29
40. A. Hassan, A. Mahmood, IEEE Access **6**, 13949 (2018). https://doi.org/10.1109/ACCESS.2018.2814818. https://ieeexplore.ieee.org/document/8314136
41. V. Ivanov, A. Sadovykh, A. Naumchev, A. Bagnato, K. Yakovlev, in *Recent Trends in Analysis of Images, Social Networks and Texts*, ed. by E. Burnaev, D.I. Ignatov, S. Ivanov, M. Khachay, O. Koltsova, A. Kutuzov, S.O. Kuznetsov, N. Loukachevitch, A. Napoli, A. Panchenko, P.M. Pardalos, J. Saramäki, A.V. Savchenko, E. Tsymbalov, E. Tutubalina. Communications in Computer and Information Science (Springer International Publishing, Cham, 2022), pp. 17–29. https://doi.org/10.1007/978-3-031-15168-2_2
42. S. Abualhaija, C. Arora, M. Sabetzadeh, L.C. Briand, E. Vaz (2019), pp. 51–62. https://doi.org/10.1109/RE.2019.00017
43. M.A. Gordon, K. Duh, N. Andrews, Compressing BERT: Studying the Effects of Weight Pruning on Transfer Learning (2020). http://arxiv.org/abs/2002.08307. arXiv:2002.08307 [cs]
44. V. Sanh, L. Debut, J. Chaumond, T. Wolf, arXiv:1910.01108 [cs] (2020). http://arxiv.org/abs/1910.01108. arXiv:1910.01108
45. J. Cleland-Huang, R. Settimi, X. Zou, P. Solc, Requir. Eng. **12**(2), 103 (2007)
46. Certification Commission for Health Information Technology (2007). https://www.cchit.org/work/criteria/, https://www.cchit.org/work/inpatient-criteria/
47. A. Rashwan, O. Ormandjieva, R. Witte, in *The 37th Annual International Computer Software & Applications Conference (COMPSAC 2013)*. IEEE (IEEE, 2013), pp. 381–386. https://doi.org/10.1109/COMPSAC.2013.64
48. OWASP Application Security Verification Standard. https://github.com/OWASP/ASVS/
49. K. Song, X. Tan, T. Qin, J. Lu, T.Y. Liu, MPNet: Masked and Permuted Pre-training for Language Understanding (2020). http://arxiv.org/abs/2004.09297. arXiv:2004.09297 [cs]
50. T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, A. Rush, in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations* (Association for Computational Linguistics, Online, 2020), pp. 38–45. https://doi.org/10.18653/v1/2020.emnlp-demos.6. https://www.aclweb.org/anthology/2020.emnlp-demos.6
51. N. Reimers, I. Gurevych, Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks (2019). http://arxiv.org/abs/1908.10084. arXiv:1908.10084 [cs]
52. R. Socher, A. Perelygin, J. Wu, J. Chuang, C.D. Manning, A.Y. Ng, C. Potts, in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing* (2013), pp. 1631–1642

53. T. Hedberg Jr., M. Helu, M. Newrock, Software requirements specification to distribute manufacturing data. Tech. Rep. NIST AMS 300-2, National Institute of Standards and Technology, Gaithersburg, MD (2017). https://doi.org/10.6028/NIST.AMS.300-2. https://nvlpubs.nist.gov/nistpubs/ams/NIST.AMS.300-2.pdf
54. A. Sadovykh, K. Iakovlev, A. Abherve, ARQAN Online Demonstrator by SOFTEAM (2022). http://arqan.softeam-rd.eu:8501/
55. Security Technical Implementation Guide (STIG) Complete List. https://www.stigviewer.com/stigs
56. I.E. Commission, others, IEC 62443: Security for Industrial Automation and Control Systems–Part 4-1: Secure Product Development Lifecycle Requirements. Tech. rep. (2018)
57. W. Wei, P.M. Barnaghi, A. Bargiela, Int. J. Commun. SIWN **3**, 76 (2008)
58. N. Reimers, Pretrained Models – Sentence-Transformers documentation. https://sbert.net/docs/pretrained_models.html
59. J. Han, J. Pei, H. Tong, *Data Mining: Concepts and Techniques* (Morgan Kaufmann, 2022)
60. R. Zhang, Operations on word vectors – Debiasing (2019). https://zhangruochi.com/Operations-on-word-vectors-Debiasing/2019/03/28/index.html
61. N. Reimers, Semantic Search – Sentence-Transformers documentation. https://sbert.net/examples/applications/semantic-search/README.html
62. I. Nigmatullin, A. Sadovykh, N. Messe, S. Ebersold, J.M. Bruel, in *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (2022), pp. 2–6. https://doi.org/10.1109/ICSTW55395.2022.00015. ISSN: 2159-4848
63. Modelio – UML/BPMN modeling tool by SOFTEAM. https://www.modeliosoft.com/en/
64. Z. Ahmed, S.C. Francis, in *2019 International Conference on Digitization (ICD)* (IEEE, 2019), pp. 178–182. https://doi.org/10.1109/ICD47981.2019.9105789. https://ieeexplore.ieee.org/abstract/document/9105789

# Chapter 3
# Security Requirements Formalization with RQCODE

**Andrey Sadovykh, Nan Messe, Ildar Nigmatullin, Sophie Ebersold, Maria Naumcheva, and Jean-Michel Bruel**

**Abstract** Security requirements vary in nature and form. These requirements may come from compliance checklists, implementation guidelines, corporate standards, and reports from organizations such as NIST, MITRE, and OWASP. Stakeholders may express additional requirements, depending on the context, to address threats and vulnerabilities as quickly as possible. Requirements are usually expressed in natural language, sometimes accompanied by tests, fixes, or descriptions of attack vectors. Analyzing, managing, verifying, validating, and tracing the requirements are therefore challenging as it relies heavily on human activity. Formalizing requirements for automated analysis and reuse can help to reduce human error-prone activities. Seamless Object-Oriented Requirement (SOOR) promotes a paradigm of multi-requirement views. In this paradigm, requirements are classes described in an object-oriented programming (OOP) language that combines representations in natural language with those in formal languages, such as LTL or Eiffel. The embedded formal language representations can provide means for validating requirements. In addition, the major advantage is that OOP supports seamless reuse of requirements classes and extensions through inheritance or associations. RQCODE is a novel approach based firstly on the implementation of SOOR in Java, and secondly on the applicability of SOOR to security requirements. This is done while providing a lightweight formalization through the associated tests that validate and strengthen system security according to the Security Technical Implementation Guide (STIG). We argue that this approach, also known as RQCODE, offers several advantages for formalizing, reusing, analyzing, and validating security requirements by automated means. In this chapter, we discuss the challenges of requirements specification in the cybersecurity domain and present our RQCODE approach.

A. Sadovykh (✉)
SOFTEAM, Ivry-sur-Seine, France
e-mail: andrey.sadovykh@softeam.fr

N. Messe · I. Nigmatullin · S. Ebersold · M. Naumcheva · J.-M. Bruel
IRIT, Toulouse, France
e-mail: Nan.Messe@irit.fr; Ildar.Nigmatullin@irit.fr; Sophie.Ebersold@irit.fr; Maria.Naumcheva@irit.fr; Jean-Michel.Bruel@irit.fr

A. Sadovykh et al. (eds.), *CyberSecurity in a DevOps Environment*,
https://doi.org/10.1007/978-3-031-42212-6_3

**Keywords** Requirement engineering · Natural language processing · Machine learning · Dataset · Classification · Semantic search · VeriDevOps

## 3.1 Introduction

### *3.1.1 Context*

Requirements engineering plays a crucial role in software development because it specifies the main goals of software applications, as well as the constraints of the environment. For example, the functionality a web browser provides to its users and the constraints on that functionality are both requirements. Requirements drive system development and deployment. They are the first step on the way from the customer's problem to a technical solution [1]. That is, requirements serve as a translation of user and business needs into an implementable and operational solution. Requirements engineering (RE) aims to define, document, and manage requirements to ensure a seamless transition from a problem space to a solution space at different stages of development. RE also supports traceability during the verification process to demonstrate that requirements are effectively implemented. Besides, RE includes various activities, such as requirements elicitation and analysis, specification, verification, and validation. Requirements elicitation defines the process of analysis and review of a set of system requirements through seeking, uncovering, acquiring, and elaborating [2]. The requirements specification is most often a structured document that gathers the sets of functional and nonfunctional requirements that must be imposed on the design and verification of the system [3]. The purpose of verification is to guarantee that a system conforms to its requirement specification. Validation should ensure that requirements define the system that the customer really wants.

RE addresses the full lifecycle of requirements, which is a significant effort. For example, it has been shown that approximately 15% of a project's effort is spent on RE activities. RE is considered to be the most important area of software engineering, since errors produced at this stage, if not detected until a later stage of software development, can be very costly [4]. Eliciting and specifying adequate requirements is therefore essential for a successful software development process.

Engineering requirements for business, system, or software applications involves much more than just functional requirements engineering. One also needs to engineer their quality requirements, especially security requirements. Security requirements are particularly critical today, when systems must adapt to a hostile digital environment [5].

Security requirements are of different nature and come in many different forms. These requirements can come from compliance checklists, implementation guidelines, corporate standards, and reports from organizations such as NIST, MITRE, and OWASP. Standardized security requirements and recommendations play an important role in the development of secure and trustworthy systems.

For example, the NIST Computer Security Handbook [6] considers requirements as technical features (e.g., access controls), assurances (e.g., background checks for system developers), or operational practices, defining security requirements in terms of features and functions. In this way, we know which objects containing the features need to be protected. It also highlights the underlying core reasons why these objects should be protected. In addition, it provides guidance on how to match and protect the functionality of the system to the security needs. Other standards also address various domains and systems layers. For example: IEC 62443 standard [7] deals with the Industrial Internet of Things (IoT) domain; Security Technical Implementation Guide (STIG) [8] provides security recommendations for a huge list of platforms at operating system and application level; Open Web Application Security Project (OWASP) [9] provides best practices and guidelines, mainly for web and mobile applications.

Stakeholders may express additional security requirements depending on the context, addressing specific and emergent threats and vulnerabilities in a timely manner.

There are several security concepts to consider when eliciting security requirements. The aim of security requirements is to protect assets, which can be anything of value to an organization or an individual user. For example, information stored in a database server is an asset. Assets may present vulnerabilities that can be exploited by attacks. This represents a potential threat to the asset. For example, if a database server does not sanitize user input, this constitutes a vulnerability. This vulnerability can be exploited by an SQL injection attack that compromises the functionality of the database server. As a result, threats can damage assets – the information stored in the server. Security requirements provide the verification methods and countermeasures to eliminate threats and protect assets. These requirements protect security properties such as confidentiality, integrity, and availability. For example, "ensure that any user input is validated by a sanitiser" is a security requirement that includes the countermeasure "user input validation," and this security requirement helps to safeguard data integrity. Countermeasures, or security controls, can take the form of either security mechanisms or security constraints.

### *3.1.2 Motivation*

Most companies rely on natural language (NL) to document their requirements, either in the form of "requirements documents" or "user stories," These requirements are extracted directly from customer documents, such as a request for proposal. They can also be the result of interviews with customers. In both cases, the requirements are specified in the domain language specific to the company and business area. This is one of the reasons why there are often communication gaps between different organizational units, especially when a requirement is specified in a very general form [10]. The abovementioned situation is a typical situation for security requirements, as a requirement specification may rely entirely on a standard such as OWASP or IEC62443, which provide very general guidelines that are difficult to verify. In addition, different stakeholders, even within the same organization, may use different vocabularies, which makes understanding requirements in NL problematic [11]. Providing well-formulated and unambiguous requirements in natural language is extremely difficult. It is necessary to increase their precision to ensure that the requirements are well understood, implemented, verified, validated, and traceable.

Redundancy of requirements is also a problem. For example, the same requirement may be represented several times by different stakeholders and in different forms. These redundant requirements may be contradictory or even mutually exclusive, i.e. they cannot exist simultaneously in the system [11]. Therefore, there is a need for an adequate way to manage customer requirements, avoiding redundancy and duplication.

In addition, stakeholders are often responsible for a specific concern or business area and do not necessarily have full technical expertise and understanding of the other problem-specific domains. This is typically the case for requirements engineers, who sometimes lack knowledge of implementation and testing, while testers lack knowledge of the requirements specification [10], making requirements verification a challenging and complicated activity. Quality assurance is a costly activity that may represent up to 80% of project costs. It is often suggested to left-shift the quality assurance and verify the system at the early development stages – as early as possible. This also concerns the requirements verification and especially for quality requirement verification [10]. This results in higher quality of the product, as well as reduced cost and time spent on removal of defects at earlier development phases.

Requirements engineering approaches need to manage the verification and also the traceability of requirements in large projects. If the mapping between requirements and test cases is not clear, it is difficult to ensure traceability. For example, if a requirement is removed, a lack of traceability makes it difficult to keep track of the tests that also need to be removed.

Security requirements are usually represented in a natural language form, sometimes accompanied by tests, fixes, or attack vector descriptions. Analyzing, managing, verifying, validating, and tracing security requirements are thus challenged since these activities heavily rely on humans being [12]. Formalizing the requirements, e.g., requirements documenting and describing in notations such as BPMN 2.0, UML, IDEF0, Event-B, etc. for their automated analysis and reuse are thus necessary to rigorously validate and verify candidate designs and implementations of these requirements and can help reduce human error-prone activities, but specific competencies in formalization are required.

Most formal approaches focus on requirements per se, not directly connected to design and implementation [1], which makes it difficult to align requirement engineering with other software engineering tasks. Requirement engineering involves upfront and detailed analysis, which can be at odds with agile software development [13] and DevOps. Continuous management of requirements is thus a nontrivial task, since not all of them are fixed at the beginning, and they may change over the course of the project [14, 15]. Besides, it is a challenge to not lose sight of the big picture during the implementation of complex requirements [14]. Nonfunctional requirements, e.g., security requirements, are often neglected in agile practice [15]. For example, user stories usually satisfy only system/product features.

It therefore appears necessary to unify the software process by allowing requirements to benefit from concepts, notations, and tools that are also applicable to other development tasks.

## 3.2 Related Work

### *3.2.1 Requirements Formalization Methods*

To make requirements precise, researchers have for years advocated the use of mathematics-based notations and methods, known as "formal." Many requirement formalization approaches exist, differing in their style, scope, and applicability.

Bruel et al. in [1] have identified five categories of approaches to specify requirements: natural language, semiformal, automata/graphs, mathematical, and seamless (programming-language-based) that are defined as follows:

1. **Natural language** approaches express requirements in English or another human language. Natural language has a significant role during requirements formalization that has proved to be crucial in the development of computerized systems. The required comprehension of system domain knowledge is ensured either via documents and text analysis or by means of stakeholder interviews. Similarly, validation of the technical specification is conducted by oral discussions and interpretation with stakeholders [16].
2. **Semiformal** approaches are based on notations that are partially formalized, e.g., SysML. They represent requirements as artifacts (such as SysML.Blocks) and connects them to other artifacts to demonstrate semantic relations such as dependency, refinement, or derivation [1]. The analysis still remains mostly manual by constructing and considering various viewpoints.
3. **Automata/graphs** methods are based on automata or graph theory. They deal with the concepts of automata, formal languages, grammar, algorithms, computability, decidability, and complexity. Most commonly the requirements are formalized as finite automata. A finite automaton is a simply idealized machine used to recognize patterns within input taken from some character set [17].
4. **Mathematical** methods are based on fundamental mathematical and algebra formalisms such as Event-B [18], alloy [19], form-L [20], VDM [21], and tabular relations [22].
5. **Seamless** methods are programming language-based methods [23], applying constraint logic and programming by contract. We provide additional details in the sections below.

One of the most important properties of a requirement is verifiability – the ability to assess a requirement's fulfilment. This property directly impacts many other properties such as understandability or clearness, correctness, absence of ambiguity, and traceability. One may go to the extreme by stating that the requirement is properly formalized when a proper verification method is assigned to it. Therefore, the formalization may be considered as an activity to define proper verification means. In the same direction, formalization of security requirements is necessary to rigorously verify and validate candidate designs and implementations against these requirements. The sections below analyze the state of the art in reusable formalization of security requirements with the focus on security verification.

#### 3.2.1.1 Formalization Through Verification

Verifiability is one of the most important properties defining quality of a requirement. Verifiability directly impacts understandability and traceability. In our approach, a correctly specified requirement has to be verifiable. Verification always assumes the presence of some specification against which the verification is performed. We thus just say "security verification patterns" when we actually mean both specification and verification. Security verification patterns are expected to contain reusable specification mechanisms for applying them to arbitrary software systems. They are also expected to contain mechanisms for their own verification against candidate designs and implementations of the specified system. By "verification" we mean both static and dynamic methods. In general, approaches to software security assurance can be categorized into two categories: (1) static approaches, which work at the implementation level, without running the system under analysis, and (2) dynamic approaches, which focus on generating and running security tests with properly generated test inputs and an appropriate oracle for assessing the test execution results. Static approaches include the following categories:

- Model checking-based approaches, which take as input a formal model of the system and model-check the desired properties against that model. These approaches require an architectural or a behavioral model of the system as input and do not require that the development phase has already started.
- Code analysis-based approaches, which work with candidate program implementations of the system. Such approaches require that the development phase has already started.

Dynamic approaches include the following categories:

- Model-based testing focuses on generating tests and their inputs based on architectural and behavioral models of the system. These approaches may facilitate test-driven development of the system if the development phase has not started yet.

- Vulnerability testing performs attacks on running applications. Vulnerability testing includes the following subcategories: (a) directly attacking the application trying to break it using known attack patterns and (b) risk-based testing, which attacks the application based on identified security risks that are specific to the given problem domain and behavioral description.

### *3.2.2 Static Verification and Security Patterns*

Konrad [24], Wassermann [25], Siveroni [26, 27], Zisman [28], Dong [29], and Ouchani [30–32] share common research interest, in the sense that they conduct model checking of UML models in one or another way. The work by Ouchani et al. [33], however, has brought to our attention since it describes an approach to formalize the requirements based on CAPEC – Common Attack Pattern Enumeration and Classification. The Common Attack Pattern Enumeration and Classification (CAPEC) effort provides a publicly available catalogue of common attack patterns that helps users understand how adversaries exploit weaknesses in applications and other cyber-enabled capabilities. Attack patterns define the challenges that an adversary may face and propose countermeasures; they are descriptions of the common attributes and approaches employed to exploit known weaknesses in cyber-enabled capabilities. They derive from the concept of design patterns [34] applied in a destructive rather than constructive context and are generated from in-depth analysis of specific real-world exploit examples. Each attack pattern captures knowledge about how specific parts of an attack are designed and executed and gives guidance on ways to mitigate the attack's impact. Attack patterns help those developing applications or administrating cyber-enabled capabilities to better understand the specific elements of an attack and how to stop them from succeeding.

In [32], Ouchani and Debbabi defined approaches to specification, verification, and quantification of security in model-based systems. The authors model both the target systems and the CAPEC patterns as SysML activity diagrams. They then compute the probabilities of a given system being vulnerable to each CAPEC pattern by submitting the resulting activity diagrams to the PRISM [35] probabilistic model checker.

Kaiya et al. [36] proposed a method for a requirements analyst to automatically acquire attack candidates against a functional requirement. This method is the first CAPEC-based method to work with requirements as inputs. One can notice that the security requirements may only be concretized after functional requirements. This is because the functional requirements specify the scope of the system. At the same time, the bias is that very often a requirements specification deals with functional requirements only, while the security aspects are specified in an extremely generic way, mostly as a need to comply with "best practices." This leads to the problem that security is addressed at the later stages of the software process.

Williams et al. [37, 38] propose an ontology-based collaborative recommender system for security requirements elicitation. The proposed approach takes use cases on input and identifies relevant CAPEC patterns. It then connects the identified CAPEC patterns with the system-specific vocabulary to construct abuse cases [39] for the system in question.

The work of Jurjens [40] proposes encoding security properties in UMLsec, an extension of UML. The resulting UMLsec specification is then submitted to AutoFocus – a CASE tool that is capable of generating test sequences. These test sequences need to be instantiated in the context of a candidate system to actually test the said system.

### *3.2.3 Dynamic Verification and Security Patterns*

Sudhodanan et al. [41] proposed a methodology in which security experts can create attack patterns from known attacks. Then they describe a security testing framework that leverages attack patterns to automatically generate test cases for security testing of multiparty web applications. This approach relies on proxy-based web security scanners to record client-server interactions and automatically detect applicability of attack patterns to the recorded interactions. Sudhodanan et al. implemented their approach on top of OWASP ZAP proxy-based web security scanner and uncovered 21 previously unknown vulnerabilities in well-known multiparty web applications.

Smith and Williams [42] developed six black box security test patterns – for (1) input validation vulnerability tests, (2) force exposure tests, (3) malicious file tests, (4) malicious use of security functions tests, (5) dangerous URL tests, and (6) audit tests. They also developed a tool called Security Test Pattern Instantiator (STPI) to help software testers instantiate security test patterns based on functional requirements. Finally, Smith and Williams conducted a user case study in which 21 graduate and 26 undergraduate students used the STPI tool to develop a black box security test plan. The study revealed that the novices' decisions were very close to the "golden standard" developed by a committee of experts.

A comprehensive review of security testing techniques by Felderer et al. [43] let us identify another conceptual cluster of pattern-based approaches – risk-based approaches. The risk-based approaches use numerical evaluations of risks' severity to define the required level of test coverage when generating test cases for the associated risks. That is to say, the higher the risk's severity is, the more coverage will be required from the test cases generated from that risk.

The DIAMONDS project (ITEA 2) has developed many relevant contributions. Schieferdecker et al. provide a general overview of the model-based testing field and map the abovementioned project on it [44]. The project is said to focus on risk-based security testing and model-based fuzzing. Grosmann et al. [45] described a tool-based iterative approach that combines the CORAS [46] approach to model-driven risk analysis with automated security testing based on patterns such as CAPEC. In every iteration of the approach, the risk analysis results are fed into the process of identifying relevant security test patterns and then instantiating these patterns into actual test cases. The testing results are then fed back into the risk analysis process and so forth. Botella et al. [47] proposed an approach that starts with risk analysis, relies on an approach similar to CORAS [46], and concludes with automated security testing of the target system. The test generation process relies on CertifyIt [48], an existing model-based testing (MBT) software. CertifyIt takes on input behavioral models of the system expressed as UML statecharts and risk-based test purposes – formalizations of vulnerability test patterns. Lebeau et al. [49] describe the fundamental principles behind model-based security testing.

## 3.3 The RQCODE Approach

### *3.3.1 Seamless Object-Oriented Requirements (SOOR)*

In the SOOR approach [23], requirements are documented as software classes which makes them verifiable and reusable. The key notions of the approach are specification drivers and semantic assertions (contracts expressed by pre- and post-conditions).

*Specification drivers* are contracted routines, expressed only in terms of their formal arguments, that serve specification purposes. Specification drivers take objects to be specified as arguments and express the effect of operations on those objects with pre- and post-conditions. The example of a specification driver for the requirement *"(REQ1) A clock tick increments current second if it is smaller than 59."*, adapted from [23] is presented in Listing 3.1.

The Eiffel code in the snippet (Listing 3.1) specifies the clock tick increment requirement by defining pre- and post-conditions that can be seen under *require* and *ensure* statements.

*Seamless Object-Oriented Requirements (SOOR)* are concrete classes capturing requirements as specification drivers. Specification drivers express formal semantics of requirements. Each specification driver is supported with a comment that captures a natural language version of the same requirement.

```
req_1 (clock: CLOCK; current_second: INTEGER)
  -- A clock tick increments current second if it is smaller than 59.
    require
        clock.second <59
        clock.second =current_second
    do
        clock.tick
    ensure
        clock.second =current_second + 1
    end
```

**Listing 3.1** REQ1 in Eiffel language

Seamless Object-Oriented Requirements serve as:

- Proof obligations, since each specification driver captures formal semantics of a requirement
- Parameterized unit tests

*Seamless Object-Oriented Requirement Templates (SOORT)* are requirements patterns captured in generic and deferred classes. Libraries of requirements templates for software components and control software temporal properties, implemented in Eiffel, are publicly available [50]. To specify SOOR according to SOORT, one needs to inherit from SOORT and replace the generic parameters with specified types.

### *3.3.2 Requirements as Code (RQCODE)*

RQCODE [51] is a novel approach to apply the Seamless Object-Oriented Requirements (SOOR) paradigm to be implemented in Java language. The RQCODE approach stands for the representation of requirements as classes that contain various representations including the textual one: requirements description in natural language as well as methods for verifying these requirements, such as an acceptance test. In this way, the traceability between a requirement and its implementation is direct and can be checked at any time through the execution of the included test. Moreover, object-oriented implementation supports easy reuse of requirements and tests by the standard means, such as inheritance, provided by the language, e.g., Java. One requirement can be an extension or a specialization of another one. Each requirement can be considered as a template for requirements of a similar kind, e.g., by initializing a requirement class with different parameters.

It should be noted that we assume that a properly specified requirement should be verifiable. RQCODE concepts (Fig. 3.1) include the *Requirement* abstract class that has a mandatory statement attribute for a textual representation of the requirement and redefines the *check()* method from the *Checkable* interface for the built-in verification of the requirement. There are three possible verification results, which are *PASS*, *FAIL*, and *INCOMPLETE*. *PASS* indicates successful result of verification execution; *FAIL* result has to be returned when the verification outputs are incorrect; *INCOMPLETE* relates to the situations when a verification could not be performed. There is also the enforceable interface for the cases where a requirement can propose a guideline to modify the environment for requirement satisfiability. The *enforce()* method returns the status in the similar form of *SUCCESS*, *FAILURE*, or *INCOMPLETE*. This enforcement mechanism is quite useful in the case of security requirements, e.g., to initiate countermeasures.

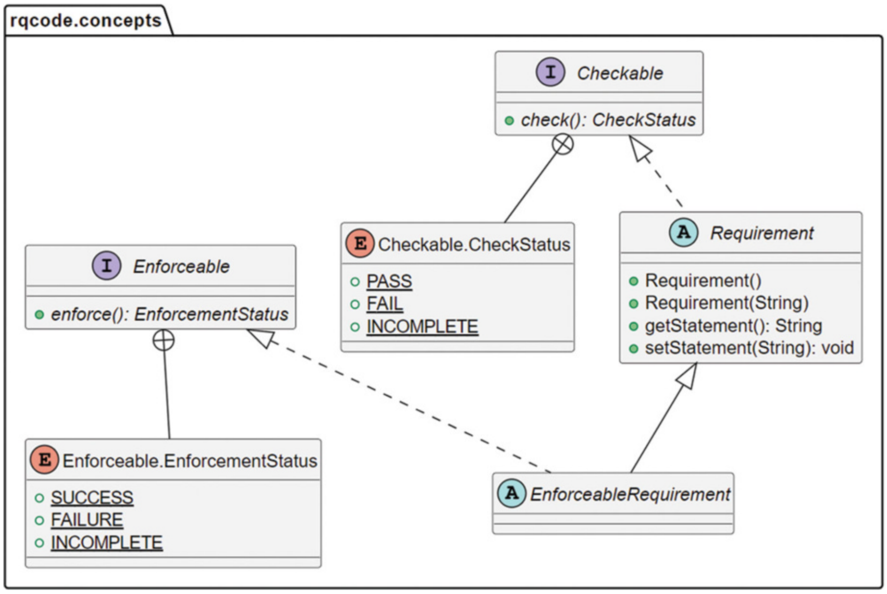


**Fig. 3.1** RQCODE concept classes (UML class diagram)

```java
public class TickIncrementRequirement extends Requirement {
   public TickIncrementRequirement() {
      super("A clock tick increments current second if it is smaller than 59");
   }
   @Override
   public CheckStatus check() {
      if (Clock.seconds < 59)
         return ((Clock.seconds + 1) == Clock.tick()) ?
         CheckStatus.PASS : CheckStatus.FAIL;
      return CheckStatus.INCOMPLETE;
   }
}
```

**Listing 3.2** REQ1 in Java language

Considering the requirement from the previous section: *"(REQ1) A clock tick increments current second if it is smaller than 59."*; the RQCODE class would be as follows in Listing 3.2.[1]

[1] More details of the example are in https://github.com/VeriDevOps/RQCODE/tree/master/src/main/java/rqcode/example

The *TickIncrementRequirement* is initialized with the statement parameter that is a textual representation of the requirement. The *Clock* is an external class that shall increment its *seconds* attribute when the *tick* operation is called. The *check()* method verifies the satisfiability of the requirement with a test. This representation of the requirement is arguably easy to read by anyone familiar with Java. This specification is verifiable by simple execution of the *check()* method. One may notice that REQ1 doesn't specify what happen if the current second is more than 59. Therefore, in the example above, the default output is *CheckStatus.INCOMPLETE*.

It is possible to reuse a requirement in this form by using Java mechanisms. In the example above, one may notice that REQ1 does not set boundaries on the value of seconds. This means that the check() will PASS in case the seconds are negative, e.g. *Clock.seconds* is equal to *-1*. We can use several ways to set up *"(REQ2) Clock seconds value must be between 0 and 59"* and combine it with REQ1. The first reuse possibility is that *TickIncrementRequirement* class can be used as a parent class to the REQ2 requirement that would extend its functionality (Listing 3.3).

This inheritance mechanism provides a direct traceability link between REQ1 and REQ2. One may clearly conclude that REQ2 is derived from the REQ1 and enhances it. In the meantime, inheritance in Java and OOP has its limitations. In Java, only one parent class is allowed. Misusing inheritance links may lead to an artificially deep inheritance tree which is commonly recognized as bad practice. This practice can lead to difficulties in understanding and maintaining the set of classes. The second possibility for reuse is that one can simply combine instances of requirement classes to combine REQ1 and REQ2 classes with the association mechanism. Imagine that REQ2 is now implemented as a separate *Requirement* class – *BoundaryRequirement*. The combination of REQ1 and REQ2 may look as follows in Listing 3.4.

```java
public class DerivedTickBoundaryRequirement extends TickIncrementRequirement {
   public DerivedTickBoundaryRequirement() {
      super.setStatement(super.getStatement() +"\nIn addition, Clock seconds
           ↪value must be between 0 and 59.");
   }
   @Override
   public CheckStatus check() {
      if (Clock.seconds > 59) return CheckStatus.FAIL;
      if (Clock.seconds < 0) return CheckStatus.FAIL;
      return super.check();
   }
}
```

**Listing 3.3** Derived REQ2 in Java language

```java
public class CombinedTickRequirement extends Requirement {
   TickIncrementRequirement tr;
   TickBoundaryRequirement br;
   public CombinedTickRequirement() {
      super("The Clock must satisfy the tick increment (REQ1) and seconds
          ↪boundary (REQ2) requirements.");
      tr = new TickIncrementRequirement();
      br = new TickBoundaryRequirement();
   }
   @Override
   public CheckStatus check() {
      if (br.check()== CheckStatus.FAIL) return CheckStatus.FAIL;
      return tr.check();
   }
}
```

**Listing 3.4** Combining REQ1 and REQ2 in Java language

The advantage of this method is that multiple requirements may be reused to form a combined one. The references to these requirements can be easily identified and navigated.

Since requirements specifications in RQCODE are Java classes, this approach inherently supports several validation mechanisms. The simplest one is a Java semantic check that is provided by the compilers, but there are also a number of static analysis methods that are integrated in Java IDEs. Moreover, object-oriented programming (OOP) analysis can be applied to the collections of security requirements. The methods presented in, e.g., [52, 52, 53], may be used to detect duplicates, circular dependencies, coupling properties, or depth of inheritance. They are relevant for analyzing if the requirements specifications are clear, atomic, non-contradicting, and verifiable. RQCODE can be applied to security requirements, and we present this possibility in the following section.

## *3.3.3 RQCODE and Temporal Requirements Patterns*

By analyzing a large number of requirements specifications for temporal properties, Dwayer [50, 54] identified that most of those temporal requirements may be mapped to very few patterns. Based on this work, we have implemented several temporal requirements patterns with the goal to cover the majority of temporal requirements kinds. The implemented patterns are listed below:

- **Eventually:** P always eventually holds.
- **Globally, Universally:** Globally, it is always the case that P holds.
- **After Q Until R Universally P:** After Q, it is always the case that P holds until R holds.
- **Globally, Universally, Response:** Globally, it is always the case that if P holds then, unless R holds, Q will eventually hold

- **Timed Globally, Universally:** Globally, it is always the case that if P held for T time units, then S holds.
- **Globally, Real-Time Response:** Globally, it is always the case that if P holds, the S eventually holds within T time units.

Figure 3.2 depicts the structure of the RQCODE temporal patterns package. In order to verify eventuality, universality, or precedence with the testing approach, we choose to implement a monitoring service in class *MonitoringLoop*. This class periodically *checks* temporal properties. The other classes implement the verification of pre-, post-, and exit conditions as required by the patterns that we listed above.

RQCODE approach supports applying those patterns to requirement specifications. For example, one may apply the *Global Universality* pattern to the *Tick Increment* requirement (REQ1).[2]
The above RQCODE example (Listing 3.5) can be translated as the following: *Globally, (REQ1) Tick increment requirement must be held for 10 seconds*. The REQ1 will be periodically *checked* according to the temporal logic of the pattern. We believe that this approach can reduce the uncertainty about temporal requirements with a practical solution based on testing.

RQCODE and temporal patterns can be applied to the security requirements that we demonstrate in the following section.

### *3.3.4 RQCODE and Security Technical Implementation Guide (STIG)*

Security Technical Implementation Guides (STIGs) [8] are a collection of guidelines for securing IT systems and products for use by the US Department of Defense (DoD) and other agencies. These guidelines provide detailed instructions on how to configure and secure various types of IT systems, including network devices, software, databases, and operating systems. The goal of the STIGs is to reduce the risk of cybersecurity threats, breaches, and intrusions by ensuring that IT systems are configured in a secure manner. These guidelines also cover:

- Database management systems
- Firewalls
- Virtualization
- Network storage
- Industrial control systems
- Email servers
- Identity and access management systems

[2] More temporal patterns in Java on Github https://github.com/VeriDevOps/RQCODE/tree/master/src/main/java/rqcode/temporal_patterns

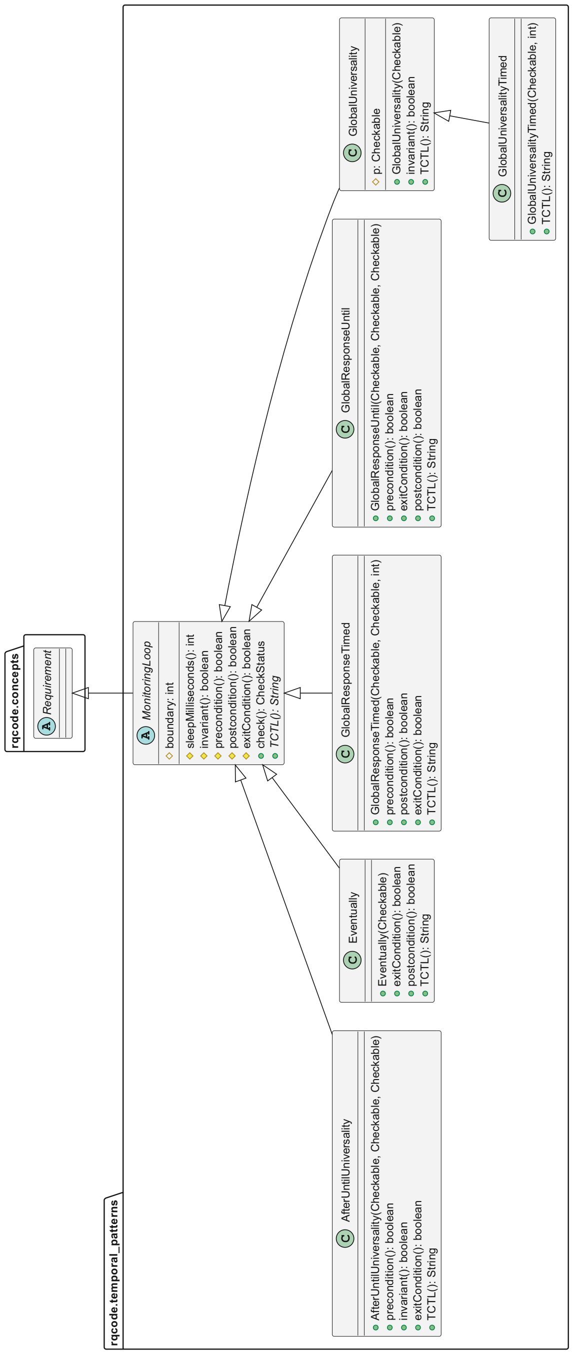


**Fig. 3.2** RQCODE implementation of temporal requirement patterns (UML class diagram)

```
    TickIncrementRequirement tr = new TickIncrementRequirement();
    ttr = new GlobalUniversalityTimed(tr, 10);
    setStatement(ttr.toString());
```

**Listing 3.5** Applying a temporal pattern to REQ1 in Java language

- Web servers
- Security information and event management systems

STIGs provide detailed guidance on how to configure and secure these types of systems, including specific settings and configurations that should be used to minimize the risk of cybersecurity threats. The guidelines are regularly updated to reflect the latest industry best practices and to address newly discovered vulnerabilities. The STIG collection consists of findings which have a specific structure. The description of each finding includes general information such as version, severity, ID, and a brief description, as well as details including two sections, Check Text and Fix Text. The Check Text section describes the conditions that the system should meet to prevent a security problem, for example, the correct configuration of an operating system. The Fix Text helps to resolve specific security issues, for example, through OS configuration.

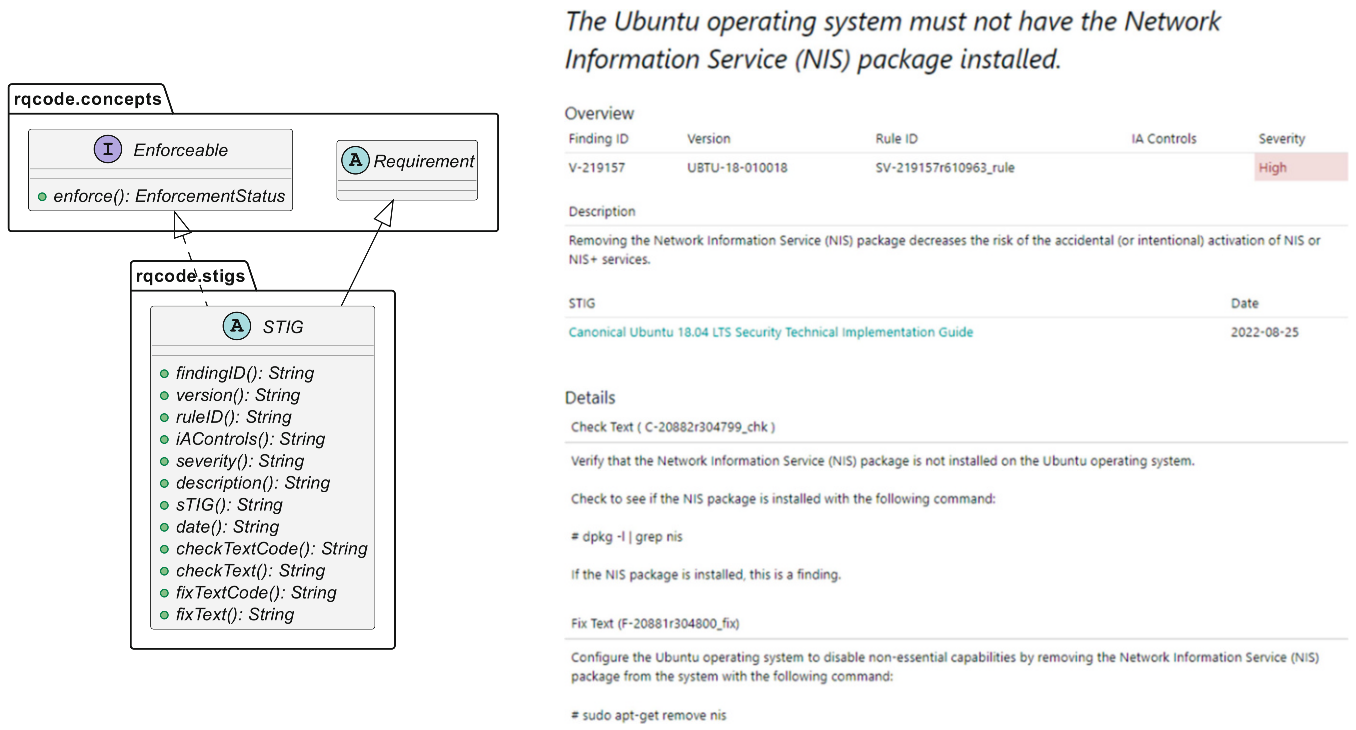
We applied the SOOR paradigm to implement STIG guidelines as RQCODE requirements (Fig. 3.3) that incorporate specific check and enforced methods to verify that the guidelines are enabled or to fix the security issue. The CheckText translates into test routines for the *check()* method, while the FixText translates into the *enforce()* method. Let us consider a STIG that provides specific recommendations for configuring and securing systems running on Ubuntu Linux. For example, STIG recommendation V_219157[3] states: *The Ubuntu operating system must not have the Network Information Service (NIS) package installed.*, since *Removing the Network Information Service (NIS) package decreases the risk of the accidental (or intentional) activation of NIS or NIS+ services*. It recommends disabling certain packages or services that are not needed for the system's intended purpose, as they could potentially introduce security vulnerabilities. In RQCODE this requirement may look as on Listing 3.6.

In the example above, one may notice the application of the "reuse by inheritance" that we introduced earlier. The *UbuntuPackagePattern* (Fig. 3.4) is a specific type of requirement that we call a pattern.

This class implements all required functionality to check the presence of a specific Ubuntu package as well as to enable or disable this package as a security enforcement measure. In addition, the *UbuntuPackagePattern* parameterizes the requirement statement text to provide the details of the STIG requirement in natural language. The STIG V_219158[4] states that *The Ubuntu operating system must not have the rsh-server package installed.* By reusing the *UbuntuPackagePattern* the RQCODE expression of this requirement may look like in Listing 3.7:

[3] https://www.stigviewer.com/stig/canonical_ubuntu_18.04_lts/2021-06-16/finding/V-219157

[4] https://www.stigviewer.com/stig/canonical_ubuntu_18.04_lts/2021-06-16/finding/V-219158



*The Ubuntu operating system must not have the Network Information Service (NIS) package installed.*

Overview

| Finding ID | Version | Rule ID | IA Controls | Severity |
|---|---|---|---|---|
| V-219157 | UBTU-18-010018 | SV-219157r610963_rule | | High |

Description

Removing the Network Information Service (NIS) package decreases the risk of the accidental (or intentional) activation of NIS or NIS+ services.

| STIG | Date |
|---|---|
| Canonical Ubuntu 18.04 LTS Security Technical Implementation Guide | 2022-08-25 |

Details

Check Text ( C-20882r304799_chk )

Verify that the Network Information Service (NIS) package is not installed on the Ubuntu operating system.

Check to see if the NIS package is installed with the following command:

```
# dpkg -l | grep nis
```

If the NIS package is installed, this is a finding.

Fix Text (F-20881r304800_fix)

Configure the Ubuntu operating system to disable non-essential capabilities by removing the Network Information Service (NIS) package from the system with the following command:

```
# sudo apt-get remove nis
```

**Fig. 3.3** STIG structure with RQCODE in Java (UML class diagram)

```java
public class V_219157 extends UbuntuPackagePattern {
   public V_219157() {
      super("nis", false);
   }
}
```

**Listing 3.6** STIG V_219157 in RQCODE by applying UbuntuPackagePattern

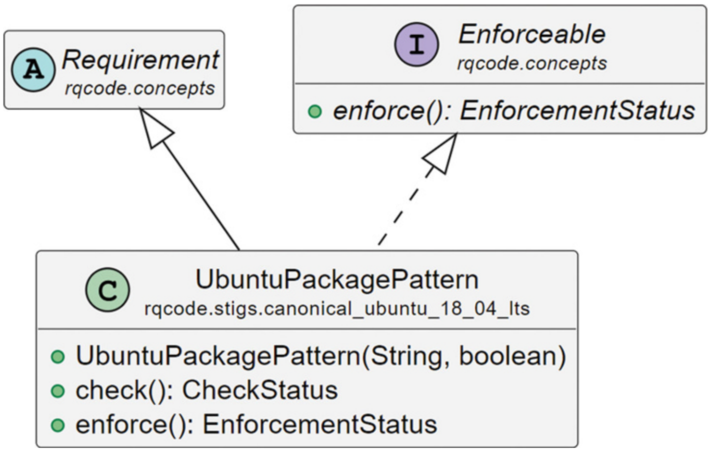

A
Requirement
rqcode.concepts
I
Enforceable
rqcode.concepts
enforce(): EnforcementStatus
C
UbuntuPackagePattern
rqcode.stigs.canonical_ubuntu_18_04_lts
UbuntuPackagePattern(String, boolean)
check(): CheckStatus
enforce(): EnforcementStatus

**Fig. 3.4** UbuntuPackagePattern class in RQCODE (UML class diagram)

```java
public class V_219158 extends UbuntuPackagePattern {
   public V_219158() {
      super("rsh-server", false);
   }
}
```

**Listing 3.7** STIG V_219158 in RQCODE by applying UbuntuPackagePattern

There are many package-related STIGs in the repository. Applying RQCODE with *UbuntuPackagePattern* supports redundancy avoidance, duplication and easy maintenance of these sets of requirements, since the fixes in this class will be propagated to all children requirements. In addition, one can notice that the rule for disabling the NIS package can be applied to many operation environments such as different versions of Ubuntu (18, 20, etc.) or even different distributions such as CentOS. To reuse the security implementation guidelines, one can generalize the *UbuntuPackagePattern* class for the CentOS as well by simply switching the package management routines from *apt* to *rpm*.

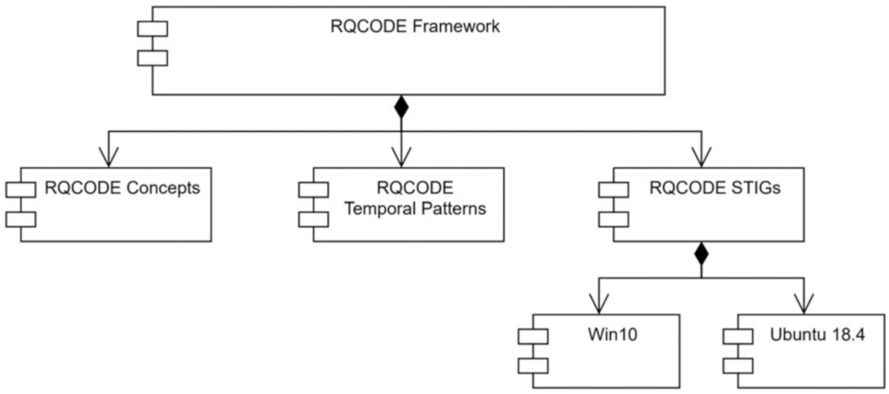


**Fig. 3.5** RQCODE framework structure (UML class diagram)

### *3.3.5 RQCODE Framework*

The structure of the RQCODE framework [55] is presented in Fig. 3.5.

The framework includes specific packages for the baseline concepts and temporal patterns since they represent the foundation. For the experimentation purposes, we prototyped examples for Windows 10 and Ubuntu 18 related STIGS. STIGS-related classes are separated in a specific package including specifics for the Windows 10 and Ubuntu 18.4 platforms. In addition, each of these platform packages includes the patterns sub-package. These pattern sub-packages contain the extracted platform specific patterns – generalized classes, such as *UbuntuPackagePattern*, that simplify the reuse of STIG requirements and guidelines.

The current framework is under evaluation by our industry partners and is subject to further refactoring, improvements, and enhancements. In particular, we are implementing a mechanism for users to indicate the need to implement additional STIGs. There is also ongoing research on integrating the RQCODE framework into DevSecOps practices.

## 3.4 Discussion

### *3.4.1 Approach for Evaluation*

RQCODE framework is currently under evaluation in the VeriDevOps Project. In this subsection we would like to discuss our approach for assessing the RQCODE performance. When designing RQCODE we intended to contribute with improvements in several RE areas:

1. Better specification – RQCODE is more formal than requirements statements in natural language, since each requirement class incorporates a verification means that can be executed on the design model or on the target system.
2. Simplified formalization – compared to several other methods based on formal logic.
3. Better traceability – since a requirement in RQCODE incorporates a verification method – generally a test case. Thus, it is completely transparent that if a verification verdict is satisfactory the requirement is fulfilled and vice versa.
4. Better reuse – since RQCODE embrace OOP, requirement is a class that can be instantiated or extended. New requirements can be created as extensions of requirements that were previously specified. All that is in accordance with OOP rules.
5. Support for analysis automation – one can assume several ways to automate the requirements analysis with RQCODE. First, the verification means can be executed to assess the requirement fulfilment for each requirement. Second, OOP analysis tools can be applied to assess the quality of requirements classes themselves. For example, by applying the depth of inheritance (DI) metrics for maintainability index metrics, one can draw an analysis for a given requirement class or a set of requirements classes.
6. Better integration with development environments and CI/CD platforms – a requirement in RQCODE is a piece of code. The industry has created a multitude of IDEs, repositories, analysis and management tools that can be directly applied. These tools are very well-known and adopted by the developers and may facilitate adoption of the RQCODE approach. Moreover, RQCODE has a natural capacity to integrate with the modern CI/CD platforms. The usage of those platforms is mainstream in quality management nowadays. The platforms help to accelerate delivery in the high pace development environment with quick releases to address the changing demand and incoming requests. This is of particular interest since new security requirements may come at any moment and have to be addressed as quickly as possible in development and in operations.

While conducting preliminary evaluation and interviews with potential users we discovered several challenges to RQCODE approach:

1. Java code is not necessarily a simple replacement for the requirements in textual form. While it may be simpler for developers who are acquainted with Java and OOP, in general, Java requires some substantial background knowledge.
2. Verifying a requirement presumes setting up a test environment. This may not necessarily be available at any moment, for example, when specifying a property of a "system to be."
3. Specifying a comprehensive way to verify a security requirement may be quite tedious or even impossible in a general case. Simpler testing approaches may result in useful but extremely partial analysis.
4. There is a need for a proper methodology and tool support that would help users to apply the RQCODE concepts and automate the analysis.

The above challenges require more analysis. At the time of writing this chapter, we concentrate on setting up a method for evaluating the potential benefits. For that we designed a multistep approach:

1. We developed a tutorial with the goal to demonstrate the benefits of the approach such as specification, traceability, and reuse.
2. Potential users are invited to complete exercises to obtain hands-on experience with RQCODE.
3. The users are invented to respond to a survey.

For the survey design, we choose the following major categories:

- Ability of RQCODE to correctly represent security requirements
- Ability of RQCODE to correctly verify the security requirement fulfilment
- Easiness to trace a requirement specified in RQCODE
- Easiness to reuse a requirement specified in RQCODE
- Easiness to analyze a requirement specified in RQCODE
- Easiness to maintain a requirement specified in RQCODE

The evaluation approach that we presented above is planned to be run with our industry partners to collect the feedback on RQCODE applicability to security requirement specification and verification.

### *3.4.2 Comparison to Other Requirements Formalization Methods*

In this section we compare RQCODE with approaches mentioned in Sect. 3.2.1) with a goal to highlight its advantages and limitations. The results used below for all methods except RQCODE are extracted from the abovementioned survey [1].

The assessed capabilities of the surveyed approaches are:

- **Formalization of functional requirements:** assess the method capability to represent system's functional requirements in a formal notation.
- **Formalization of nonfunctional requirements:** assess the method possibility of specifying nonfunctional requirements in a formal notation.
- **Validation/verification:** assess how the approach evaluates the system's response to different inputs. Validation stands for checking systems behavior against user's or customers' needs. Verification represents a capability to assess the system's compliance to the specification.
- **Traceability:** the ability to establish and control the links among requirements, specifications, design, code, and other artifacts.
- **Reuse:** the capability to create a new requirement specification as a copy or an extension of an existing one.

- **Maintenance:** over time requirements change or get refined, the trace links created, deleted, or updated. The maintenance capability indicates the facility of a method to handle the updates in the existing requirement specification.
- **Tool support:** the tool support is often cited as a barrier to adoption of a method. This property indicates to which extent a method is covered by specific tools, e.g., requirements elicitation, verification, validation, and management.
- **Learning barrier:** addresses the expected level of maturity and technical expertise of the users.
- **Analysis:** is the ability of a method to support requirements analysis activities such as determining whether a set of requirements is clear, complete, unduplicated, concise, valid, consistent and unambiguous, and resolving any apparent conflicts.
- **Developer friendliness:** the developers are the major stakeholders in the system implementation process; they have to clearly understand the requirements and the ways these requirements are verified and validated. The developer friendliness indicates the ability of a method to simplify handling of a requirement by a developer.

The ability of a method to support the abovementioned capabilities is evaluated based on the following scale:

- "+ + +" – Full matching (90–100%)
- "++" – Matches with minor deviation (70–90%)
- "+" – Matches partially (50–69%)
- "+/−" – Has few matches (0–49%)
- "−" – Negative match

We provide below a discussion of the approaches and their properties as compared to RQCODE (Table 3.1):

**Natural language** is the most used method to specify the requirements on all the levels of system specification – from business goals to inputs and outputs of specific functions. The greatest advantage of the natural language is flexibility and the highest expressive power to specify the requirements in the most natural way. There exist several formalization enforcement approaches using specific lexical structure (MoSCoW rules [56]), ontologies and templates, and guidelines for seminatural language specification (Given-When-Then [57], (T) EARS) [58, 59]. Unfortunately, as has been pointed out in many papers, the formalization power of natural language is low, leading to errors of omission and commission due to misunderstanding and misinterpretation of requirements. Natural language requirements can be handled using many tools such as DOORS, REUSE, and ticketing systems such as JIRA. These tools also provide the ability to manually specify relationships between requirements, e.g., using hyperlinks. However, support for automating analysis, verification, and validation is rather poor.

**Semiformal** approaches such as SysML[60] and EAST ADL [61] or ArchiMate [62] provide notation with defined semantics with the goal to specify various aspects of a system. SysML and EAST ADL provide means to define the high-

**Table 3.1** Comparison of RQCODE with other categories of approaches (*: new introduced capability)

| Capability\approach | RQCODE | Natural language (e.g., Textual) | Automata/graphs (e.g., FSP/LTSA) | Semiformal (e.g., SysML) | Mathematical (e.g., Event-B) |
|---|---|---|---|---|---|
| Formalization of functional requirements | + | - | +++ | + | +++ |
| Formalization of nonfunctional requirements | + | +/- | - | +/- | - |
| Validation/ Verification | ++ | - | ++ | +/- | +++ |
| Traceability | ++ | - | + | +/- | |
| Reuse | +++ | - | + | + | ++ |
| Maintenance | +++ | - | + | +/- | - |
| Tool support | +++ | +++ | ++ | ++ | + |
| Learning barrier* | + | - | ++ | + | ++ |
| Analysis* | ++ | - | +++ | - | - |
| Developer friendliness* | ++ | +++ | + | - | - |

level components of the system and to specify operational rules and constraints. The architectural element can be linked to the requirement objects. The requirements are expressed in natural language, but the relationships between requirements can have specific meanings such as “refinement” or positive and negative “influence.” The constraints can be specified in a mathematical language as a function of parameters specified in the model. There are several methods for analyzing these models, for example, by mapping them to frameworks such as MATLAB. However, several challenges remain: (1) the model and the traceability links have to be maintained manually, (2) there are certain concerns about the maintainability of large-scale models, and (3) the approaches are aimed at system architects and are less developer-friendly.

**Mathematical** approaches, e.g., Isabelle [63], B [64], Event-B [18], etc., are capable of formalizing functional requirements since they rely on mathematical formalisms. For example, Event-B is a formal modelling and verification approach that can model the static and dynamic parts of a system using contexts and machines based on refinements. Requirements represented in Event-B can be verified using the proof obligations associated with a model and its refinements, each of which includes contexts and machines. The expression in each level of refinement must be proven to be consistent with its higher level. Therefore, this category of approaches is capable of verifying and validating requirements. As Even-B is based on refinement, high-level models can be reused in lower levels, and Even-B models can also be reused by other relevant projects. It is also tool-supported. Requirement engineers can use the Rodin environment in which they can define Event-b models, refine models, and verify with proof obligations. However, users are expected to have some knowledge of how to use Rodin and define models. Current mathematical approaches are efficient in formalizing, verifying, and validating functional requirements, which define specific behavior or functions but are less efficient in dealing with nonfunctional requirements, which specify criteria by which the operation of a system can be judged.

**Automata/graph-based** approaches (e.g., LTL [65], problem frames [66] (Jackson 2005) and FSP/LTSA [67]), support the formalization of functional requirements. These approaches are based on the mathematical theories of graphs and automata and are supported by effective graphical representations. Thus, despite their mathematical foundations, they are easy to use (they are particularly popular with students). These approaches are mainly dedicated to dynamic representations and can be complemented with formal notations that allow them to be validated and verified, but if this is true for LTSA, for example, (safety), this is not the case for problem frames that do not provide any verification way.

**RQCODE**, compared to the other methods, has several advantages. In particular, it is considered to be developer friendly as it is implemented as a Java framework. So developers who are familiar with unit testing – presumably any Java developer – will be able to use RQCODE. Requirements in RQCODE are source files, which are handled naturally in version control repositories such as GitHub, and test automation is applied using continuous integration pipelines. Reuse methods are common in Java development and are therefore natural in RQCODE. There are a number of automated tools for static and dynamic analysis of Java code. These tools can be applied to RQCODE classes representing requirements to provide an analysis of the RQCODE specification in terms of, for example, maintainability. Expressing requirements in Java also has its limitations. Java is a programming language that has its rules and requires quite concrete instructions to obtain compilable code. Compared to natural language, the expressiveness and flexibility of RQCODE is lower, while there is a need to learn and use a general purpose programming language. However, compared to more formal methods, the learning barrier should be lower, while the number of tools for editing, handling, and managing RQCODE classes should facilitate the adoption of this approach.

## 3.5 Conclusions

In this paper we have outlined the current approaches for requirement engineering that provide formalized means for verification. We claim that these methods are not necessarily developer-friendly as they require expert knowledge and lack tool support. To cope with these limitations, we propose the RQCODE method which suggests using the Java programming language to express requirements within classes that incorporate verification means, e.g., tests. We argue that this method enhances reusability and traceability for requirements specifications. We illustrate how RQCODE can be applied to the security domain and in particular with the set of requirements from the Security Technical Implementation Guide. We illustrate the reuse mechanisms with several examples. The work on RQCODE is currently ongoing, and we prepare the evaluation stage where we plan to gather feedback from industry partners. In particular we plan to run a dedicated tutorial that targets to demonstrate the presumed benefits of the RQCODE approach. At the end of the tutorial, we plan to gather the feedback in a form of a survey that analyses industry perceptions. The source code of the RQCODE framework and the STIG examples is publicly available [55]. We plan to further refactor, maintain, and enhance this project. We are interested in comparing this approach with test-driven development and analyze the RQCODE usage within the continuous integration and delivery paradigm. Another area of research is the DevSecOps practices, where RQCODE requirements related to security may be located, imported, or reused with the goal to automate security requirements analysis and verification.

**Acknowledgments** This work is partially supported by the VeriDevOps [68] project funded by the Horizon 2020 program under the grant agreement No. 957212 (VeriDevOps project).

## References

1. J.M. Bruel, S. Ebersold, F. Galinier, M. Mazzara, A. Naumchev, B. Meyer, ACM Comput. Surv. **54**(5), 93:1 (2021). https://doi.org/10.1145/3448975
2. D. Zowghi, C. Coulin, in *Engineering and Managing Software Requirements*, ed. by A. Aurum, C. Wohlin (Springer, Berlin/Heidelberg, 2005), pp. 19–46. https://doi.org/10.1007/3-540-28244-0_2
3. IEEE 830-1993, *IEEE Recommended Practice for Software Requirements Specifications*. https://standards.ieee.org/ieee/830/1221/
4. A. Chakraborty, M.K. Baowaly, A. Arefin, A.N. Bahar, J. Emerg. Trends Comput. Inf. Sci. **3**(5) (2012)
5. D. Firesmith, et al., J. Object Technol. **2**(1), 53 (2003)
6. B. Guttman, E. Roback, NIST (1995). https://www.nist.gov/publications/introduction-computer-security-nist-handbook. Last Modified: 2018-11-10T10:11-05:00 Publisher: Barbara Guttman, E Roback
7. I.E. Commission, others, IEC 62443: Security for Industrial Automation and Control Systems–Part 4-1: Secure Product Development Lifecycle Requirements. Tech. rep. (2018)
8. Security Technical Implementation Guide (STIG) Complete List. https://www.stigviewer.com/stigs
9. OWASP Web Security Testing Guide | OWASP Foundation. https://owasp.org/www-project-web-security-testing-guide/
10. G. Sabaliauskaite, A. Loconsole, E. Engström, M. Unterkalmsteiner, B. Regnell, P. Runeson, T. Gorschek, R. Feldt, in *Requirements Engineering: Foundation for Software Quality*, ed. by R. Wieringa, A. Persson. Lecture Notes in Computer Science (Springer, Berlin/Heidelberg, 2010), pp. 128–142. https://doi.org/10.1007/978-3-642-14192-8_14
11. L. Karlsson, Å.G. Dahlstedt, B. Regnell, J.N. och Dag, A. Persson, Inf. Softw. Technol. **49**(6), 588 (2007)
12. D. Cuddeback, A. Dekhtyar, J. Hayes, in *2010 18th IEEE International Requirements Engineering Conference* (2010), pp. 231–240. https://doi.org/10.1109/RE.2010.35. ISSN: 2332-6441
13. R. Kasauli, G. Liebel, E. Knauss, S. Gopakumar, B. Kanagwa, in *2017 IEEE 25th International Requirements Engineering Conference (RE)* (IEEE, 2017), pp. 352–361
14. E.M. Schön, D. Winter, M.J. Escalona, J. Thomaschewski, in *International Conference on Agile Software Development* (Springer, Cham, 2017), pp. 37–51
15. I. Inayat, S.S. Salim, S. Marczak, M. Daneva, S. Shamshirband, Comput. Hum. Behav. **51**, 915 (2015)
16. C. Rolland, C. Proix (2006), pp. 257–277. https://doi.org/10.1007/BFb0035136
17. Finite Automata. https://www.cs.rochester.edu/u/nelson/courses/csc_173/fa/fa.html
18. J.R. Abrial, *Modeling in Event-B: System and Software Engineering* (Cambridge University Press, 2010). Google-Books-ID: 23UgAwAAQBAJ
19. D. Jackson, *Software Abstractions, Revised Edition: Logic, Language, and Analysis* (MIT Press, 2011)
20. D. Bouskela, A. Falcone, A. Garro, A. Jardin, M. Otter, N. Thuy, A. Tundis, Requir. Eng. **27**(1), 1 (2022). https://doi.org/10.1007/s00766-021-00359-z
21. D. Bjørner, in *Mathematical Studies of Information Processing*, ed. by E.K. Blum, M. Paul, S. Takasu. Lecture Notes in Computer Science (Springer, Berlin/Heidelberg, 1979), pp. 326–359. https://doi.org/10.1007/3-540-09541-1_33
22. D.L. Parnas, in *The Future of Software Engineering*, ed. by S. Nanz (Springer, Berlin/Heidelberg, 2011), pp. 125–148. https://doi.org/10.1007/978-3-642-15187-3_8
23. A. Naumchev, B. Meyer, Computer Languages, Systems & Structures **49**, 119 (2017). https://doi.org/https://doi.org/10.1016/j.cl.2017.04.001. https://www.sciencedirect.com/science/article/pii/S1477842416301981

24. S. Konrad, B.H. Cheng, L.A. Campbell, R. Wassermann, *Requirements Engineering for High Assurance Systems (RHAS'03)*, vol. 11 (2003). https://documentcloud.adobe.com/link/review?uri=urn:aaid:scds:US:df310295-34de-492a-b389-5359146eed19
25. R. Wassermann, B.H. Cheng, in *Michigan State University, PLoP Conference on Citeseer* (Citeseer, 2003). https://documentcloud.adobe.com/link/review?uri=urn:aaid:scds:US:562417be-c44d-47d7-a9c5-bc366f207ca9
26. I. Siveroni, A. Zisman, G. Spanoudakis, in *2008 Third International Conference on Availability, Reliability and Security* (IEEE, 2008), pp. 96–103. https://documentcloud.adobe.com/link/review?uri=urn:aaid:scds:US:9b2d1580-441d-4aef-8914-2b6cd4e3b3d1
27. I. Siveroni, A. Zisman, G. Spanoudakis, Requir. Eng. **15**(1), 95 (2010). https://doi.org/10.1007/s00766-009-0091-y
28. A. Zisman, in *Second International Conference on Internet and Web Applications and Services (ICIW'07)* (IEEE, 2007), pp. 8–8
29. J. Dong, T. Peng, Y. Zhao, Inf. Softw. Technol. **52**(3), 274 (2010)
30. S. Ouchani, O.A. Mohamed, M. Debbabi, M. Pourzandi, in *Software Engineering Research, Management and Applications 2010* (Springer, 2010), pp. 163–177
31. S. Ouchani, O.A. Mohamed, M. Debbabi, in *2013 IEEE 7th International Conference on Software Security and Reliability* (IEEE, 2013), pp. 227–236. https://documentcloud.adobe.com/link/review?uri=urn:aaid:scds:US:f71fe199-99f8-4c77-bd6d-2122f84d3cec
32. S. Ouchani, M. Debbabi, Computing **97**(7), 691 (2015)
33. S. Ouchani, Y. Jarraya, O.A. Mohamed, in *2011 Ninth Annual International Conference on Privacy, Security and Trust* (IEEE, 2011), pp. 142–149. https://documentcloud.adobe.com/link/review?uri=urn:aaid:scds:US:2f653129-e94b-46e0-a9e5-55614133ab3b
34. R.C. Martin, Object Mentor **1**(34), 597 (2000)
35. M. Kwiatkowska, G. Norman, D. Parker, in *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation* (Springer, 2002), pp. 200–204
36. H. Kaiya, S. Kono, S. Ogata, T. Okubo, N. Yoshioka, H. Washizaki, K. Kaijiri, in *International Conference on Advanced Information Systems Engineering* (Springer, 2014), pp. 343–348
37. I. Williams, X. Yuan, in *2017 IEEE Cybersecurity Development (SecDev)* (IEEE, 2017), pp. 85–86
38. I. Williams, in *2018 IEEE 26th International Requirements Engineering Conference (RE)* (IEEE, 2018), pp. 448–453
39. G. McGraw, IEEE Secur. Privacy **2**(2), 80 (2004)
40. J. Jürjens, in *International Conference on The Unified Modeling Language* (Springer, 2002), pp. 412–425
41. A. Sudhodanan, A. Armando, R. Carbone, L. Compagna, others, in *NDSS* (2016)
42. B. Smith, L. Williams, in *2012 IEEE Sixth International Conference on Software Security and Reliability* (IEEE, 2012), pp. 108–117
43. M. Felderer, M. Büchler, M. Johns, A.D. Brucker, R. Breu, A. Pretschner, in *Advances in Computers*, vol. 101, ed. by A. Memon (Elsevier, 2016), pp. 1–51. https://doi.org/10.1016/bs.adcom.2015.11.003. https://www.sciencedirect.com/science/article/pii/S0065245815000649
44. I. Schieferdecker, J. Grossmann, M. Schneider, arXiv preprint arXiv:1202.6118 (2012)
45. J. Großmann, M. Schneider, J. Viehmann, M.F. Wendland, in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation* (Springer, 2014), pp. 322–336
46. M.S. Lund, B. Solhaug, K. Stølen, *Model-Driven Risk Analysis: the CORAS Approach* (Springer Science & Business Media, 2010)
47. J. Botella, B. Legeard, F. Peureux, A. Vernotte, in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation* (Springer, 2014), pp. 337–352. https://documentcloud.adobe.com/link/review?uri=urn:aaid:scds:US:decdf192-2e7b-45cf-b6fe-0c517eb8b764
48. F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, in *Proceedings of the 3rd International Workshop on Automation of Software Test* (2008), pp. 45–48

49. F. Lebeau, B. Legeard, F. Peureux, A. Vernotte, in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops* (IEEE, 2013), pp. 445–452. https://documentcloud.adobe.com/link/review?uri=urn:aaid:scds:US:9769a8fb-23f9-4eec-a146-da36d95333a8
50. A. Naumchev, Requirements templates in Eiffel (2021). https://github.com/anaumchev/requirements_templates. Original-date: 2018-08-04T06:58:02Z
51. K. Ismaeel, A. Naumchev, A. Sadovykh, D. Truscan, E.P. Enoiu, C. Seceleanu, in *2021 IEEE 29th International Requirements Engineering Conference Workshops (REW)* (2021), pp. 357–363. https://doi.org/10.1109/REW53955.2021.00063
52. S. Chidamber, C. Kemerer, IEEE Trans. Softw. Eng. **20**(6), 476 (1994). https://doi.org/10.1109/32.295895. Conference Name: IEEE Transactions on Software Engineering
53. G. Succi, W. Pedrycz, S. Djokic, P. Zuliani, B. Russo, Empir. Softw. Eng. **10**(1), 81 (2005). https://doi.org/10.1023/B:EMSE.0000048324.12188.a2
54. M.B. Dwyer, G.S. Avrunin, J.C. Corbett, in *Proceedings of the 21st International Conference on Software Engineering* (1999), pp. 411–420
55. A. Sadovykh, Rqcode framework on github (2022). https://github.com/VeriDevOps/RQCODE
56. E. Miranda, in *Agile Processes in Software Engineering and Extreme Programming*, ed. by V. Stray, K.J. Stol, M. Paasivaara, P. Kruchten (Springer International Publishing, Cham, 2022), Lecture Notes in Business Information Processing, pp. 19–34. https://doi.org/10.1007/978-3-031-08169-9_2
57. J. Smart, *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle* (Simon and Schuster, 2014)
58. A. Mavin, P. Wilkinson, in *2010 18th IEEE International Requirements Engineering Conference* (2010), pp. 277–282. https://doi.org/10.1109/RE.2010.39. ISSN: 2332-6441
59. D. Flemström, H. Jonsson, E.P. Enoiu, W. Afzal, in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)* (2021), pp. 351–361. https://doi.org/10.1109/ICST49551.2021.00047. ISSN: 2159-4848
60. S. Friedenthal, A. Moore, R. Steiner, *A Practical Guide to SysML: The Systems Modeling Language* (Morgan Kaufmann, 2014)
61. V. Debruyne, F. Simonot-Lion, Y. Trinquet, in *Architecture Description Languages*, ed. by P. Dissaux, M. Filali-Amine, P. Michel, F. Vernadat (Springer US, Boston, MA, 2005), IFIP The International Federation for Information Processing, pp. 181–195. https://doi.org/10.1007/0-387-24590-1_12
62. M.M. Lankhorst, H.A. Proper, H. Jonkers, in *Enterprise, Business-Process and Information Systems Modeling*, ed. by T. Halpin, J. Krogstie, S. Nurcan, E. Proper, R. Schmidt, P. Soffer, R. Ukor. Lecture Notes in Business Information Processing (Springer, Berlin/Heidelberg, 2009), pp. 367–380. https://doi.org/10.1007/978-3-642-01862-6_30
63. M. Strecker, in *International Conference on Automated Deduction* (Springer, 2002), pp. 63–77
64. K. Lano, *The B Language and Method: A Guide to Practical Formal Development* (Springer Science & Business Media, 2012). Google-Books-ID: aoPuBwAAQBAJ
65. A. Bauer, M. Leucker, C. Schallhart, ACM Trans. Softw. Eng. Methodol. **20**(4), 14:1 (2011). https://doi.org/10.1145/2000799.2000800
66. M. Jackson, Inf. Softw. Technol. **47**(14), 903 (2005). https://doi.org/10.1016/j.infsof.2005.08.004. https://www.sciencedirect.com/science/article/pii/S0950584905001229
67. H. Foster, S. Uchitel, J. Magee, J. Kramer, in *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06 (Association for Computing Machinery, New York, NY, USA, 2006), pp. 771–774. https://doi.org/10.1145/1134285.1134408
68. A. Sadovykh, G. Widforss, D. Truscan, E.P. Enoiu, W. Mallouli, R. Iglesias, A. Bagnto, O. Hendel, in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)* (2021), pp. 1330–1333. https://doi.org/10.23919/DATE51398.2021.9474185. ISSN: 1558-1101

# Part II
# Prevention at Development Time

# Chapter 4
# Vulnerability Detection and Response: Current Status and New Approaches

Ángel Longueira-Romero, Rosa Iglesias, Jose Luis Flores, and Iñaki Garitano

**Abstract** The rapid evolution of industrial components, the paradigm of Industry 4.0, and the new connectivity features introduced by 5G technology all increase the likelihood of cybersecurity incidents. These incidents have to be managed to limit or mitigate their impact, and in most cases, they are a consequence of existing vulnerabilities. This scenario raises the need for a tool that enables a faster (tracking the vulnerability state over time) and more precise (detect root cause) response. The defined Extended Dependency Graph (EDG) model is capable to respond to this need, being able to analyze known vulnerabilities for a given device over time. The EDG model can be applied throughout the entire lifespan of a device to track vulnerabilities, identify new requirements, root causes, and test cases. It also helps prioritize patching activities. This chapter defines the key terms used in vulnerability analysis, as well as the current state of the art of vulnerability analysis in both scientific literature and standards. The EDG model is described in more depth together with its fundamental elements: (1) the directed graph representation of the internal structure of the device, (2) the set of quantitative metrics based on the Common Vulnerability Scoring System (CVSS), and (3) the algorithm to build the EDG for a given device.

**Keywords** Cybersecurity · Industrial components · Embedded systems · Vulnerability analysis · Vulnerability detection · Quantitative metrics · CVSS

Supported by Ikerlan Technology Research Center, Basque Research and Technology Alliance (BRTA).

Á. Longueira-Romero (✉) · R. Iglesias · J.L. Flores
Ikerlan Technology Research Centre, Arrasate, Spain
e-mail: alongueira@ikerlan.es; riglesias@ikerlan.es; jlflores@ikerlan.es

I. Garitano
Mondragon Unibertsitatea, Arrasate, Spain
e-mail: igaritano@mondragon.edu

A. Sadovykh et al. (eds.), *CyberSecurity in a DevOps Environment*,
https://doi.org/10.1007/978-3-031-42212-6_4

## 4.1 Introduction

Embedded systems (ES) are the driving force of almost every industrial field, such as automotive, energy production, and transportation [12, 21, 22, 29, 68]. These types of components are rapidly evolving [55, 63] and increasing in number [70]. This increasing is related to several factors: (1) The reuse of open-source hardware and software (2) New connectivity features (3) More complex systems

Open-source hardware and software, and commercial off-the-shelf (COTS) components are being integrated to speed up their development [25, 65, 66]. COTS are easy to use, but they can introduce vulnerabilities, creating potential entry points for attackers [46, 77].

Industrial components are providing more advanced connectivity features, enabling new automation applications, services, and data exchange. This new connectivity, boost by the fifth generation (5G) of wireless technology for cellular networks, will further open the window of exposure to any threat [13, 22, 44, 70].

The complexity of industrial systems is also increasing with the integration of new trends, such as the Internet of Things (IoT) [8, 13, 18, 23], cloud computing, artificial intelligence (AI) [18, 75], and big data. The extensive use of these technologies further opens the windows for attackers [9, 15, 37, 45, 71, 73]. Complexity is a critical aspect of industrial components design, because it is closely related to the number of vulnerabilities [1, 47].

This scenario point security is a key aspect of ESs. Moreover, numerous attacks have been reported targeting industrial enterprises across the globe since 2010 [36]. An exponential rise in such attacks is predicted for future years [20, 64].

In summary, the rapid evolution of ESs, their connectivity, and the integration of more and more features increase their attack surface. This makes it essential to protect their use in environments such as critical infrastructures [60, 69]. The sophistication of attacks, a larger attack surface, and the ease of attacks thanks to exploits and tools that decrease the necessary knowledge of the attackers highlight the need to invest more in cybersecurity. The numerous attacks targeting industrial enterprises across the globe since 2010 reinforce this fact [36], and an exponential rise in such attacks is predicted for the upcoming years [20, 64]. As a consequence, security is turning into a critical issue for ESs [19]. However, security by itself is not enough, and the degree of coverage of the implemented countermeasures also has to be evaluated to know whether they are sufficient. Tracking the security status of an ES [5, 35] and considering both software and hardware in the evaluation would also be desirable [6, 29, 42, 74].

## 4.2 Background

In this section, the basic concepts related to embedded systems, cybersecurity, and vulnerability analysis are presented.

- **Embedded Device**: Special purpose device designed to directly monitor or control an industrial process (e.g., PLCs, and wireless field sensor devices) [30, 32]. Typical attributes of these devices are limited storage, limited number of exposed services, programmed through an external interface, embedded operating systems (OSs), or firmware equivalent, real-time scheduler, may have an attached control panel, and may have a communications interface.
- **System Under Test (SUT):** Any system or component that is the objective of any kind of evaluation [11].
- **Vulnerability analysis:** Systematic examination of an information system or product to determine the adequacy of security measures, identify security deficiencies, provide data from which to predict the effectiveness of proposed security measures, and confirm the adequacy of such measures after implementation, including the identification and characterization of potential security vulnerabilities [31, 59].
- **Weakness:** Weaknesses are flaws, faults, bugs, and other errors in software and hardware design, architecture, code, or implementation that, if left unaddressed, could result in systems, networks, and hardware being vulnerable to attacks [53] (e.g., buffer overflow).
- **Vulnerability:** Flaw in a software, firmware, hardware, or service component resulting from a weakness that can be exploited, causing a negative impact to the confidentiality, integrity, or availability of an impacted component or components [51]. Vulnerabilities can be classified as both known and unknown [7]. In some cases, unknown vulnerabilities might be known for a group of attackers that do not want to disclose their knowledge to take malicious advantages of it (zero-day vulnerabilities) [24].
- **Attack Pattern:** An attack pattern is a description of the common attributes and approaches employed by adversaries to exploit known weaknesses in cyber-enabled capabilities [49]. Attack patterns define the challenges that an adversary may face and how they go about solving it.
- **Common Platform Enumeration (CPE) Scheme:** Naming scheme[1] for describing and identifying applications, operating systems, software, and hardware, including industrial control systems, such as supervisory control and data acquisition (SCADA) [43, 57]. CPE is operated by the NIST [61]. The latest version at the time this chapter was written is version 2.3.
- **Common Weakness Enumeration (CWE):** Community-developed list of common software and hardware weakness types, each one associated with some CVEs (explained in the next subsection) [53, 54]. CWE is operated by the MITRE Corporation [48]. The latest version at the time this chapter was written is version 4.3.
- **Common Vulnerabilities and Exposures (CVE):** List of common identifiers for publicly known cybersecurity vulnerabilities [52, 53, 58] operated by the MITRE

[1] Version 8.0.6001 of Internet Explorer for its *beta* update can be represented using version 2.3 of the CPE as *cpe:2.3:a:microsoft:internet_explorer:8.0.6001:beta:*:*:*:*:*:**

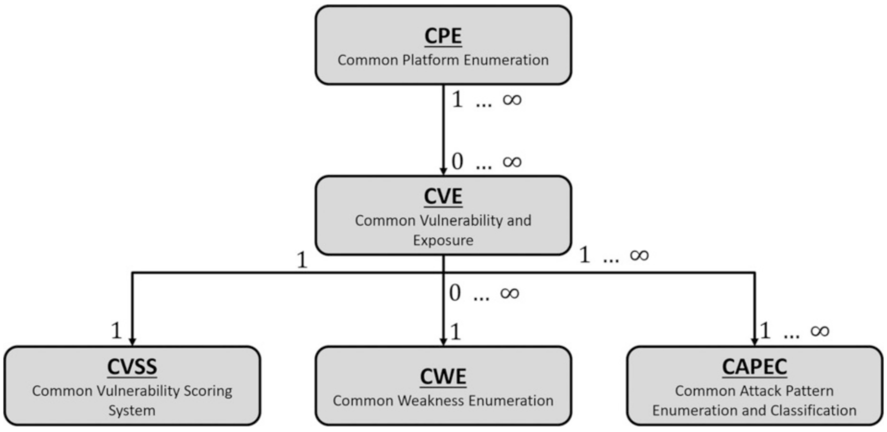

CPE
Common Platform Enumeration
1 ... ∞
0 ... ∞
CVE
Common Vulnerability and Exposure
1
1 ... ∞
0 ... ∞
1
1
1 ... ∞
CVSS
Common Vulnerability Scoring System
CWE
Common Weakness Enumeration
CAPEC
Common Attack Pattern Enumeration and Classification

**Fig. 4.1** Relationship between the security standards defined by MITRE and NIST. (Taken from [16])

Corporation [48]. Each CVE includes its severity [16]. The latest CVE version is always available in its official site.[2]

- **Common Vulnerability Scoring System (CVSS):** Public framework that provides a standardized method for assigning quantitative values (scores) to security vulnerabilities (CVE) [58] according to their severity [17]. A CVSS score is a decimal number in the range [0.0, 10.0]. The latest version at the time this chapter was written is version 3.1.
- **Common Attack Pattern Enumeration and Classification (CAPEC):** Comprehensive dictionary that provides a publicly available classification taxonomy of known attack patterns (security threats) [50]. CAPEC utilizes a qualitative approach, rating both likelihood and impact in a five-step value scale ranging from very low to very high. Finally, each CAPEC records the weaknesses (CWEs) that the attack pattern can exploit. The latest version at the time this chapter was written is version 3.4.

Figure 4.1 shows the relationship between the different standards [16].

## 4.3 State of the Art of Vulnerability Analysis in ESs

This section presents the current status of vulnerability analysis, both in the literature and current standards.

[2] https://cve.mitre.org/

### 4.3.1 *Vulnerability Analysis in Security Standards*

This review is focused on how standards conduct vulnerability analysis, the use of metrics, their management of the life cycle of the device, the techniques that they propose, and the security evaluation of both software and hardware.

#### 4.3.1.1 ISA/IEC 62443

The ISA/IEC 62443-4-1 technical document [31] specifies that vulnerabilities in a product should be identified and characterized, including known and unknown vulnerabilities [7, 24]. Two requirements in the standard are related to vulnerability analysis [31]:

- **Requirement SVV-3. Vulnerability Testing** Perform tests that focus on identifying and characterizing potential and known security vulnerabilities in the product (i.e., fuzz testing, or black box known vulnerability scanning).
- **Requirement SVV-4. Penetration Testing** Identify and characterize security-related issues, focusing on discovering and exploiting security vulnerabilities in the product (i.e., penetration testing).

#### 4.3.1.2 Common Criteria

The CC defines five tasks in the Vulnerability Assessment class [14]:

1. Vulnerability survey
2. Vulnerability analysis
3. Focused vulnerability analysis
4. Methodical vulnerability analysis
5. Advanced methodical vulnerability analysis

Every task checks for the presence of publicly known vulnerabilities. Penetration testing is also performed. The main difference among the five levels of vulnerability analysis described here is the deepness of the analysis of known vulnerabilities and the penetration testing.

### 4.3.2 *Vulnerability Analysis in the Literature*

In this subsection, the most relevant works related to vulnerability analysis are reviewed.

Vulnerability analysis efforts are mainly focused on computer networks. For this reason, most of the current research is based on directed graphs.

Homer et al. [26] presented a quantitative model for computer networks that objectively measures the likelihood of a vulnerability. Attack graphs and individual vulnerability metrics, such as CVSS, and probabilistic reasoning are applied to produce a sound risk measurement.

Zhang et al. [27, 76] developed a quantitative model that can be used to aggregate vulnerability metrics in an enterprise network based on attack graphs.

George et al. [20] propose a graph-based model to address the security issues in Industrial IoT (IIoT) networks. It represents the relationships among entities and their vulnerabilities, serving as a security framework for the risk assessment of the network. Risk mitigation strategies are also proposed.

Poolsappasit et al. [67] propose a risk management framework using Bayesian networks that enables a system administrator to quantify the chances of network compromise at various levels.

Muñoz-González et al. [56] propose the use of efficient algorithms to make an exact inference in Bayesian attack graphs, which enables static and dynamic network risk assessments. This model is able to compute the likelihood of a vulnerability and can be extended to include zero-day vulnerabilities, attacker's capabilities, or dependencies between vulnerability types.

Hu et al. [28] Hu et al. propose a network security risk assessment method that is based on the improved hidden Markov model (I-HMM).

Longueira-Romero et al. [38] proposed an Extended Dependency Graph (EDG) model that performs continuous vulnerability assessment to determine the source and nature of vulnerabilities and enhance security throughout the entire life cycle of industrial components. The proposal is built on a directed graph-based structure and a set of metrics based on globally accepted security standards.

## 4.4 Vulnerability Analysis Approaches: Analyzing Extended Dependency Graphs (EDG)

In this section, we review in more detail the Extended Dependency Graphs (EDGs) approach for vulnerability analysis in ESs. EDGs are intended to:

- Identify the root causes and nature of vulnerabilities, which will enable the extraction of new requirements and test cases.
- Extract new requirements and test cases.
- Support the prioritization of patching.
- Track vulnerabilities during the whole lifespan of industrial components.
- Support the development and maintenance of industrial components.

To accomplish this task, the EDG model comprises two basic elements: (1) the model itself, which is capable of representing the internal structure of the system under test, and (2) a set of metrics, which allow conclusions to be drawn about the origin, distribution, and severity of vulnerabilities. Both the model and metrics are very flexible and exhibit some properties that make them suitable for industrial components and can also be applied to enhance the ISA/IEC 62443 standard.

The content in this section is distributed into three subsections, namely:

1. **Model:** The EDG model is explained, together with the systems in which it can be applied and the algorithms that are used to build it.
2. **Metrics:** Metrics are a great tool to measure the state of the system and to track its evolution. The proposed metrics and their usage are described in this section.
3. **Properties:** The main features of the EDG model and metrics (e.g., granularity of the analysis, analysis over time, and patching policy prioritization support) are described in detail.

### *4.4.1 Description of the Model*

The EDG model is based on directed graphs. It requires knowledge of the internal structure of the device to be evaluated (i.e., the assets, both hardware and software, that comprise it and the relationships between them). This section defines the most basic elements that make up the model, the algorithms to build it for any give system, and its graphical representation.

**Definition 4.1** A system under test (SUT) is now represented by an Extended Dependency Graph (EDG) model $G = (\langle A, V\rangle, E)$ that is based on directed graphs, where $A$ and $V$ represent the nodes of the graphs, and $E$ represents its edges or dependencies:

- $A = \{a_1, \dots, a_n\}$ represents the set of assets in which the SUT can be decomposed, where $n$ is the total number of obtained assets. An asset $a$ is any component of the SUT that supports information-related activities and includes both hardware and software [4, 34, 41]. Each asset is characterized by its corresponding Common Platform Enumeration (CPE) [10, 43, 57] identifier, while its weaknesses are characterized by the corresponding CWE identifier. In the EDG model, the assets are represented by three types of nodes in the directed graphs (i.e., root nodes, asset nodes, and cluster).
- $V = \{v_1, \dots, v_q\}$ represents the set of known vulnerabilities that are present in each asset of $A$, where $q$ is the total number of vulnerabilities. They are characterized by the corresponding CVE and CVSS values. In the EDG model, vulnerabilities are represented using two types of nodes in the directed graphs (i.e., known vulnerability nodes and clusters).
- $E = \{e_{ij} | \forall i, j \in \{1, \dots, n + q\}$ such that $i \neq j\}$ represents the set of edges or dependencies among the assets and between assets and vulnerabilities. $e_{ij}$ indicates that a dependency relation is established from asset $a_i$ to asset $a_j$. Dependencies are represented using two different types of edges in the EDG (i.e., normal dependency and deprecated asset/updated vulnerability edges).

In other words, the EDG model can represent a system, from its assets to its vulnerabilities, and dependencies as a directed graph. Assets and vulnerabilities are represented as nodes, whose dependencies are represented as arcs in the graph. The information in the EDG is further enhanced by introducing metrics.

The EDG model of a given SUT will include four types of node and two types of dependencies. The graphical representation for each element is shown in Table 4.1. Figure 4.2 shows an example of a simple EDG and its basic elements. All of the elements that make up an EDG will be explained in more detail below.

**Table 4.1** Overview of the information that is necessary to define each of the EDG elements

| Symbol | Notation | Meaning | Values |
|---|---|---|---|
| □ | $A(t)$ | root Node/ Device node | $CPE_{current}$ |
| ○ | $a(t)$ | Asset node | $CPE_{previous}, CPE_{current}, CWE_{a_i}(t)$ |
| ◌ | $\underline{a}(t)$ | Cluster | $\{CPE_{previous}, CPE_{current}, CWE_{a_i}(t)\}$, $\{CVE_{a_i}(t), CVSS_{v_i}(t), CAPEC_{w_i}(t)\}$, $\{Dependencies\}$ |
| ▼ | $v(t)$ | Known vulnerability | $CVE_{a_i}(t), CVSS_{v_i}(t), CAPEC_{w_i}(t)$ |
| → | $e(t)$ | Dependency relation | – |
| ⇢ | $e(t)$ | Updated asset/patched vulnerability | – |

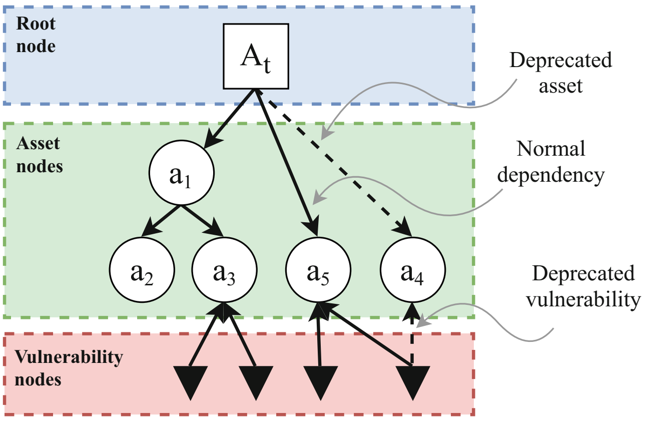


**Fig. 4.2** Basic elements of an EDG. Note that clusters are not displayed in this figure. For clusters, see Fig. 4.4. For metrics definition, see Sect. 4.5

### *4.4.2 Types of Node*

The EDG model uses four types of node:

- **Root nodes** represent the SUT.
- **Asset nodes** represent each one of the assets of the SUT.
- **Known vulnerability nodes** represent the vulnerabilities in the SUT.
- **Clusters** summarize the information in a subgraph.

**Root nodes** (collectively, set $G_R$) are a special type of node that represent the whole SUT. Any EDG starts in a root node and each EDG will only have one single root node, with an associated timestamp ($t$) that indicates when the last check for changes was done. This timestamp is formatted following the structure defined in the ISO 8601 standard for date and time [33].

**Asset nodes** (collectively, set $G_A$) represent the assets that comprise the SUT. The EDG model does not impose any restrictions on the minimum number of assets that the graph must have. However, the SUT can be better monitored over time when there are a higher number of assets. Moreover, the results and conclusions obtained will be much more accurate. Nevertheless, each EDG will have as many asset nodes as necessary, and the decomposition of assets can go as far and to as low level as needed.

Each asset node will be characterized by the following set of values:

- $CPE_{current}$: Current value for the CPE. This points to the current version of the asset it refers to.
- $CPE_{previous}$: Value of the CPE that identifies the previous version of this asset. This will be used by the model to trace back all the versions of the same asset over time, from the current version to the very first version.
- $CWE_{a_i}(t)$: Set of all the weaknesses that are related to the vulnerabilities present in the asset. The content of this list can vary depending on the version of the asset.

Figure 4.3 illustrates how the tracking of the versions of an asset using CPE works. On the one hand, version $a_i$ is the current version of asset $a$. It contains its current CPE value and the CPE of its previous version. On the other hand, $a_2$ and $a_1$ are previous versions of asset $a$. The last value of $a_1$ points to a null value. This indicates that it is the last value in the chain and therefore the very first version of the asset $a$.

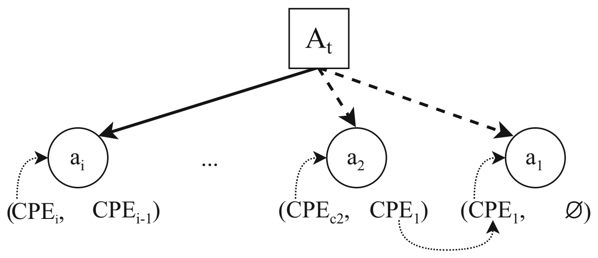


**Fig. 4.3** Tracking dependencies between the previous and current CPE values for asset *a*

**Known vulnerability nodes** (collectively, set $G_V$) represent a known vulnerability present in the asset that it relates to. Each asset will have a known vulnerability node for each known vulnerability belonging to that asset. Assets alone cannot tell how severe or dangerous the vulnerabilities might be, so unique characterization of vulnerabilities is crucial [20].

To identify each known vulnerability node, each will be characterized by the following set of features (formally defined in Sect. 4.5:

- $CVE_{a_i}(t)$: This serves as the identifier of a vulnerability of asset $a_i$.
- $CVSS_{v_i}(t)$: This metric assigns a numeric value to the severity of vulnerability $v_i$. Each CVE has a corresponding CVSS value.
- $CAPEC_{w_i}(t)$: Each vulnerability (CVE) is a materialization of a weakness (CWE) $w_i$ that can be exploited using a concrete attack pattern. In many cases, each CWE has more than one Common Attack Pattern Enumeration and Classification (CAPEC) [49, 50] associated. Consequently, this field is a set that contains all the possible attack patterns that can exploit the vulnerability that is being analyzed.

**Clusters** (collectively, set $G_S$) are a special type of node that summarizes and simplifies the information contained in a subgraph in an EDG. Figure 4.4 shows how the clusters work.

To identify each cluster and to be able to recover the information that they summarize, each is characterized by the data that define each of the elements that they contain: $\{CPE_{previous}, CPE_{current}, CWE_{a_i}(t)\}$, $\big(CVE_{a_i}(t), CVSS_{v_i}(t), \{CAPEC_{w_i}(t)\}\big)$, and their dependencies.

Two types of criteria can be used to create clusters and to simplify the obtained graph (Fig. 4.4):

1. **Absence of vulnerabilities**: Using this criterion, clusters will group all nodes that contain no associated vulnerabilities.
2. **CVSS score below a certain threshold**: With this criterion, a threshold for the CVSS scores will be chosen. Nodes whose CVSS score is less than the defined threshold will be grouped into a cluster.

It is worth noticing that applying the second set of criteria (establishing a CVSS threshold) will always return a graph that is at least as simple or as complex as the one that would be obtained using the absence of vulnerability criterion. In the best case, the graph will be simpler. This is because both criteria treat assets with no vulnerabilities in the same way, so those will always be simplified. On the other hand, establishing a CVSS threshold allows the model for further simplifications.

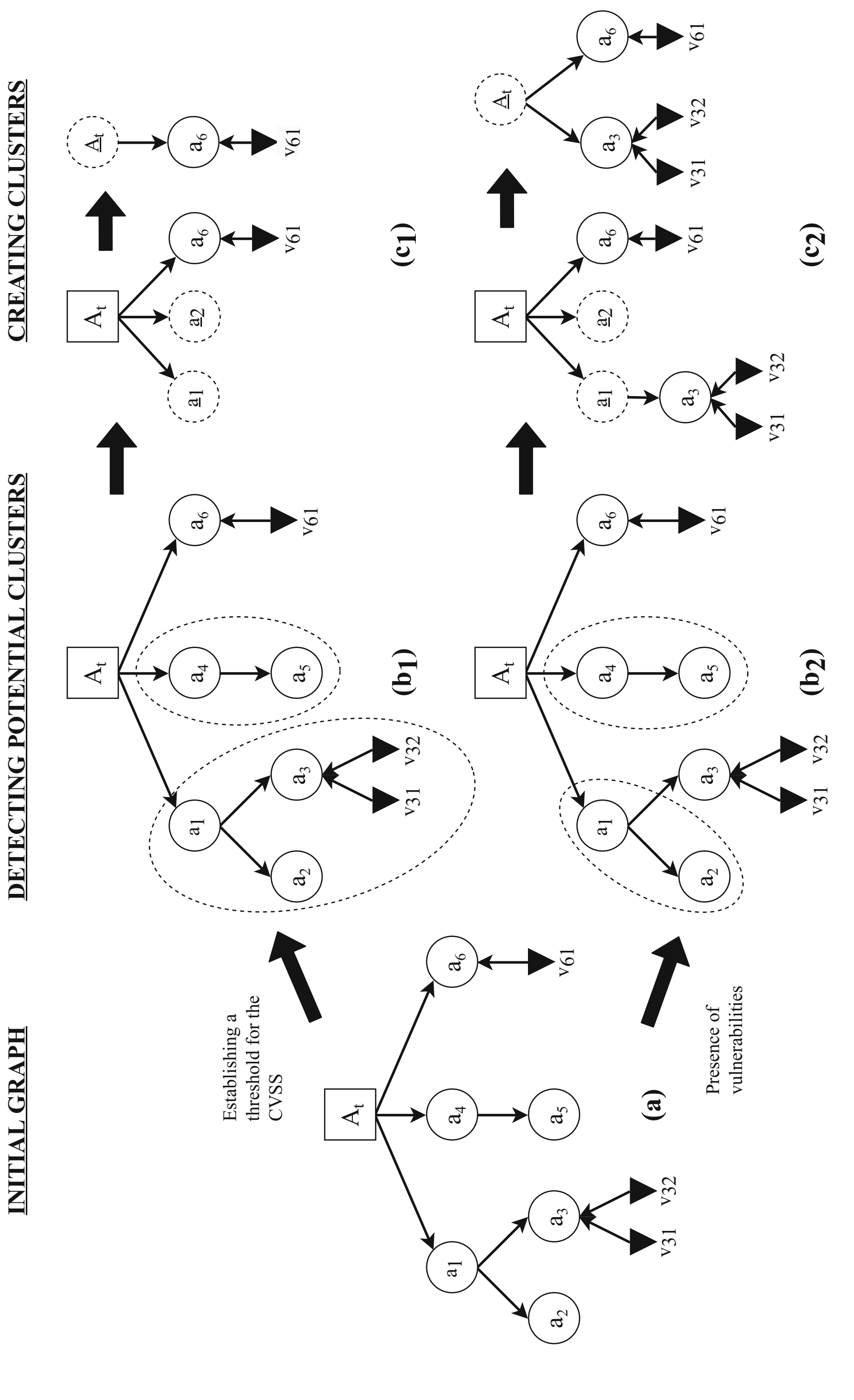

INITIAL GRAPH
DETECTING POTENTIAL CLUSTERS
CREATING CLUSTERS
Establishing a threshold for the CVSS
Presence of vulnerabilities
(a)
(b1)
(b2)
(c1)
(c2)
A_t
a1
a2
a3
a4
a5
a6
v31
v32
v61

**Fig. 4.4** Creating clusters. Application of the two proposed criteria to the creation of clusters to simplify the graph: (1) Establishing a threshold to select which vulnerability stays outside the cluster (upper side). (2) Choosing the absence of vulnerability as the criterion to create clusters (lower side). The severity value (CVSS) for $v_{31}$ and $v_{32}$ is supposed to be lower than the establish threshold

### 4.4.3 Types of Edge

In the EDG model, edges play a key role representing dependencies. Two types of edge can be identified:

- **Normal dependencies** relate two assets, or an asset and a vulnerability. They represent that the destination element depends on the source element. Collectively, they are known as set $G_D$.
- **Deprecated asset or patched vulnerability dependencies** indicate when an asset or a vulnerability is updated or patched. They represent that the destination element used to depend on the source element. Collectively, they are known as set $G_U$.

The possibility of representing old dependencies brings the opportunity to reflect the evolution of the SUT over time. When a new version of an asset is released, or a vulnerability is patched, the model will be updated. Their dependencies will change then from a normal dependency to a deprecated asset or vulnerability dependency to reflect that change.

### 4.4.4 Steps to Build the Model

This section explains the process and algorithms that were used to build the corresponding EDG of a given SUT. The main scenarios that can be found are also described.

Before extracting useful information about the SUT, the directed graph associated with the SUT has to be built. This comprises several steps, which are described in the following paragraphs (see the flowchart in Fig. 4.5a, b):

**Step 1 – Decompose the SUT into assets** For the model to work properly, it relies on the SUT being able to be decomposed into assets. With this in mind, the first step involves obtaining the assets of the SUT, either software or hardware. In the CC, this process is called modular decomposition of the SUT [11]. Ideally, every asset should be represented in the decomposition process, but this is not compulsory for the model to work properly. Each one of the assets obtained in this step will be represented as an asset node. In this step, the dependencies among the obtained assets are also added as normal dependencies.

**Step 2 – Assign a CPE to each asset** Once the assets and their dependencies have been identified, the next task is to assign the corresponding CPE identifier to each asset. If there is no publicly available information of a certain asset and, therefore, it does not have a CPE identifier, then it is always possible to generate one using the fields described in the CPE naming specification documents [10] for internal use in the model.

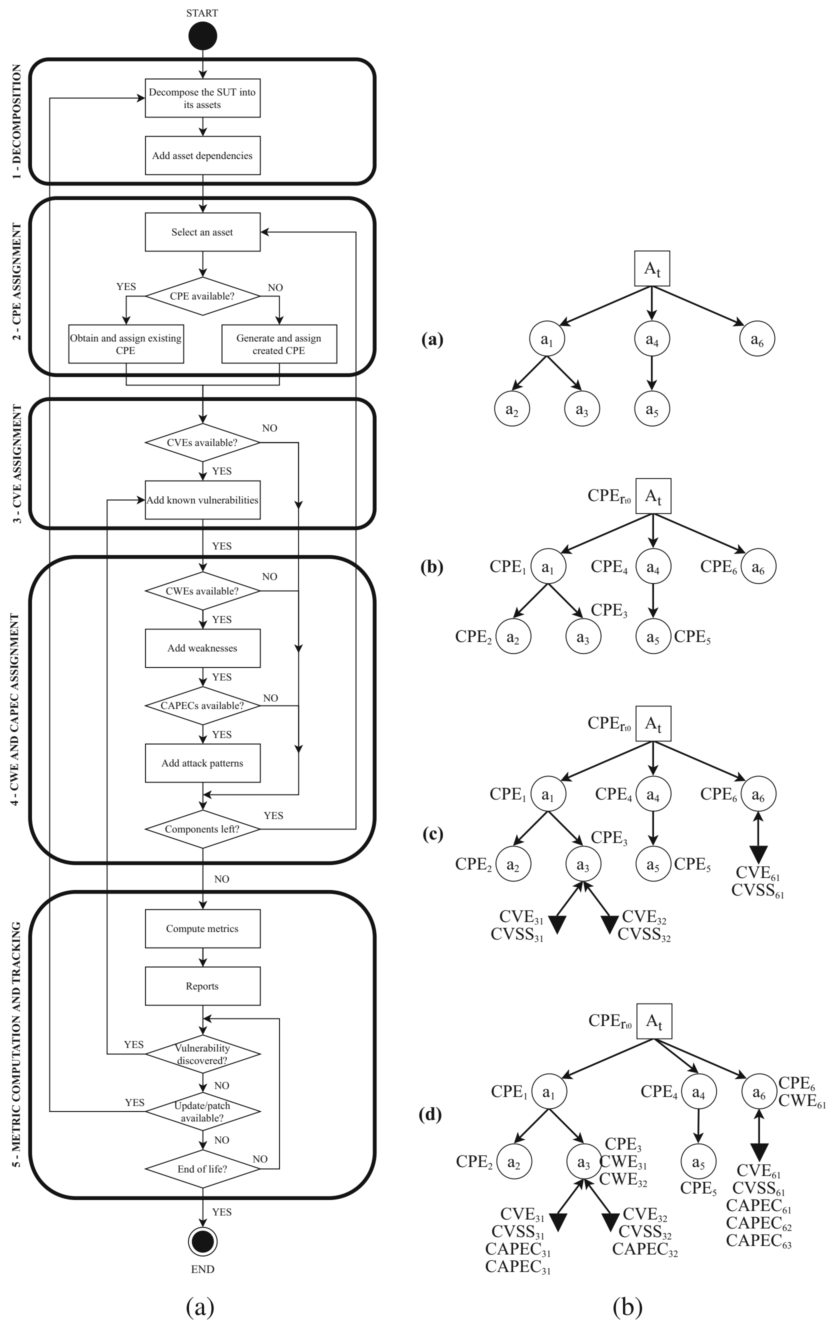


**Fig. 4.5** Three simple graphs. (**a**) Algorithm to generate the initial EDG of a given SUT. (**b**) Example of the process of building the EDG model of a given SUT *A*

**Step 3 – Add known vulnerabilities to the assets** In this step, the vulnerabilities ($CVE_{a_i}(t)$) of each asset are set. This is done by consulting public databases of known vulnerabilities [52, 62] looking for existing vulnerabilities for each asset. When a vulnerability is found, it is added to the model of the SUT, including its dependencies. If there were no known vulnerabilities in an asset, then the asset would become the last leaf of its branch. In this step, the corresponding value of the CVSS of each vulnerability is also added to the model.

**Step 4 – Assign to each asset its weaknesses and possible CAPECs** After the vulnerabilities, the corresponding weaknesses to each vulnerability ($CWE_{a_i}(t)$) are added, along with the corresponding attack patterns ($CAPEC_{w_i}(t)$) for each weakness. If there is no known vulnerability in an asset, then there will be no weaknesses. Meanwhile, it would be possible to have a known vulnerability in an asset, but no known weakness or attack pattern for that vulnerability. Finally, more than one CAPEC can be assigned to the same weakness. Consequently, it would be common to have a set of possible CAPECs that can be used to exploit the same weakness. It is worth noting that not all of them could be applied in every scenario.

**Step 5 – Computing metrics and tracking the SUT** At this point, the EDG of the SUT is completed with all the public information that can be gathered. This last step is to calculate the metrics defined (for further information, see Sect. 4.5.), generating the corresponding reports,and tracking the state of the SUT for possible updates in the information of the model. This step is always triggered when the SUT is updated. This can imply that a new asset can appear, an old asset can disappear, an old vulnerability can be patched, or a new one can appear in the SUT. All of these scenarios will be reflected in the model as they arise during its life cycle.

## 4.5 Security Metrics

The EDG model that was reviewed in the previous sections is by itself capable of representing the internal structure of the SUT, and it can display it graphically for the user. This representation not only includes the internal assets of the SUT but it also captures their relationships, existing vulnerabilities, and weaknesses. Moreover, assets, vulnerabilities, and weaknesses are easily identified using their corresponding CPE, CVE, and CWE values, respectively. All together, this constitutes a plethora of information that the model can use to improve the development and maintenance steps of the SUT, enhance its security, and track its status during its whole life cycle. Metrics are a great tool to integrate these features into the model.

Metrics can serve as a tool to manage security, make decisions, and compare results over time. They can also be used to systematically improve the security level of an industrial component or to predict its security level in a future point in time.

In this section, the basic definitions that serve as the foundation of the metrics are described. Then, the corresponding metrics are introduced to complement the functionality of the EDG model. The main feature of these metrics is that they all depend on time as a variable, so it is possible to capture the actual state of the SUT, track its evolution over time, and compare the results.

### *4.5.1 Basic Definitions*

In this section, the basic concepts on which the definitions of the metrics will be based are formalized.

**Definition 4.2** The set of all possible weaknesses at a time $t$ is represented as $CWE(t)$, where

$$CWE(t) = \{cwe_1, \ldots, cwe_m\} \tag{4.1}$$

and $m$ is the total number of weaknesses at time t. This set contains the whole CWE database defined by MITRE [54].

**Definition 4.3** The set of all of the possible vulnerabilities at a time $t$ is represented as $CVE(t)$ where

$$CVE(t) = \{cve_1, \ldots, cve_p\} \tag{4.2}$$

and $p$ is the total number of vulnerabilities. This set contains the whole CVE database defined by MITRE [52].

**Definition 4.4** The set of all possible attack patterns at a time $t$ is represented as $CAPEC(t)$, where

$$CAPEC(t) = \{capec_1, \ldots, capec_q\} \tag{4.3}$$

and $q$ is the total number of attack patterns at time t. This set contains the whole CAPEC database defined by MITRE [50].

**Definition 4.5** The set of weaknesses of an asset $a_i$ at a time $t$ is defined as

$$\begin{aligned} CWE_{a_i}(t) = \{cwe_j | cwe_j \text{ is in the asset } a_i \text{ at time } t \wedge cwe_j \in CWE(t) \\ \wedge \forall k \neq j, cwe_j \neq cwe_k\} \end{aligned} \tag{4.4}$$

From this expression, the set of all the weaknesses of a particular asset throughout its life cycle is defined as

$$CWE_{a_i} = \bigcup_{t=1}^{T} CWE_{a_i}(t) \tag{4.5}$$

where $|CWE_{a_i}|$ is the total number of non-repeated weaknesses in its entire life cycle.

**Definition 4.6** The set of vulnerabilities of an asset $a_i$ at a time $t$ is defined as

$$CVE_{a_i}(t) = \{cve_j | cve_j \text{ is in the asset } a_i \text{ at time } t \wedge cve_j \in CVE(t)\} \tag{4.6}$$

From this expression, the set of vulnerabilities of an asset throughout its entire life cycle is defined as

$$CVE_{a_i} = \bigcup_{t=1}^{T} CVE_{a_i}(t) \tag{4.7}$$

where $|CVE_{a_i}|$ is the total number of vulnerabilities in its entire life cycle.

**Definition 4.7** The set of weaknesses of a SUT $A$ with $n$ assets at a time $t$ is defined as

$$CWE_A(t) = \bigcup_{i=1}^{n} CWE_{a_i}(t) \tag{4.8}$$

**Definition 4.8** The set of vulnerabilities of a SUT $A$ with $n$ assets at a time $t$ is defined as

$$CVE_A(t) = \bigcup_{i=1}^{n} CVE_{a_i}(t) \tag{4.9}$$

**Definition 4.9** The set of vulnerabilities associated to the weakness $cwe_j$ and to the asset $a_i$ at a time $t$ is defined as

$$CVE_{a_i|cwe_j}(t) = \{cve_k | cve_k \text{ associated to weakness } cwe_j \text{ and to asset } a_i \text{ at time } t\} \tag{4.10}$$

It is worth noting that CWE is used as a classification mechanism that differentiates CVEs by the type of vulnerability that they represent. A vulnerability will usually have only one associated weakness, and weaknesses can have one or more associated vulnerabilities [16].

**Definition 4.10** The partition $j$ of an asset $a_i$ at time $t$ conditioned by a weakness $cwe_k$ is defined as

$$CVE_{a_i|cwe_k}(t) = \{cwe_l | cwe_l = cwe_k \wedge cwe_l \in CVE_{a_i}(t)\} \tag{4.11}$$

**Definition 4.11** The partition $j$ of the SUT $A$ at time $t$ conditioned by a weakness $cwe_k$ is defined as

$$CVE_{A|cwe_k}(t) = \{cwe_l | cwe_l = cwe_k \wedge cwe_l \in CVE_A(t)\} \tag{4.12}$$

**Definition 4.12** The set of attack patterns associated to a weakness $w_i$ at a time $t$ is defined as

$$\begin{aligned} CAPEC_{w_i}(t) = \{capec_j | capec_j \text{ can exploit weakness } w_i \text{ at time} \\ t \wedge capec_j \in CAPEC(t)\} \end{aligned} \tag{4.13}$$

.

**Definition 4.13** The set of metrics that are defined in this research work based on the EDG model is defined as

$$M = \{m_1, \ldots, m_r\} \tag{4.14}$$

where $r$ is the total number of metrics. This set can be extended, defining more metrics according to the nature of the SUT.

## 4.5.2 Metrics

This section will describe the metrics that were defined based on the EDG model and the previous definitions. Although it might seem trivial, the most interesting feature of these metrics is that they all depend on time. Using time as an input variable for the computation of the metrics opens the opportunity to track results over time, compare them, and analyze the evolution of the status of the SUT. Furthermore, some metrics take advantage of time to generate an accumulated value, giving information about the life cycle of the SUT. Table 4.2 shows all of the defined metrics, their definition, and their reference values.

In addition to the metrics in Table 4.2, the model allows the definition of other types of metrics according to the analysis to be performed, and the nature of the SUT (e.g., the vulnerability evolution function for SUT $A$ up to time $t$ for all vulnerabilities could be defined as the linear regression of the total number of vulnerabilities in each time $t$ for SUT $A$, or using any other statistical model).

## 4.5.3 Properties

Together, the EDG model and the defined metrics exhibit a series of characteristics that make them suitable for vulnerability assessment.

**Table 4.2** Defined metrics for the EDG model

| Metric | | Definition | Reference value |
|---|---|---|---|
| Vulnerabilities | $M_0(A) = \frac{\lvert CVE_A(t)\rvert}{n(t)}$ | Arithmetic mean of vulnerabilities in the SUT $A$, where $n(t)$ is the number of assets in a SUT at a time $t$. $M_0$ shows how many vulnerabilities would be present in each asset if they were evenly distributed among the assets of the SUT. The result of $M_0$ can serve as a preliminary analysis of the SUT, related to the criticality of its state. From Eq. 4.8 | $M_0 < 1$: The number of vulnerabilities is lower than the number of assets. $M_0 \geq 1$: Every asset has at least one vulnerability |
| | $M_1(A, t) = \lvert CVE_A(t)\rvert$ | Number of vulnerabilities in a SUT $A$ at time $t$. From Eq. 4.8 | Ideally, the values of $M_1$ should be zero (no vulnerability in $A$), but the lower the value of $M_1$, the better |
| | $M_2(A) = \sum_{t=1}^{T} \lvert CVE_A(t)\rvert = \sum_{t=1}^{T} M_1(A, t)$ | Number of vulnerabilities in a SUT $A$ throughout its entire life cycle $T$. This metric computes the accumulated value of the number of vulnerabilities of a SUT throughout its entire life cycle. From Eq. 4.8 | The lower the value of $M_2$, the better. |
| | $M_3(a_i, t) = \lvert CVE_{a_i}(t)\rvert$ | Number of vulnerabilities in an asset $a_k$ at time $t$ The values of $M_3$ can be useful during a vulnerability analysis, or when performing a penetration test, to identify the asset with more vulnerabilities. From Eq. 4.6 | Ideally, the value of $M_3$ should be zero |

(continued)

**Table 4.2** (continued)

| | Metric | Definition | Reference value |
|---|---|---|---|
| | $M_4(a_k, t) = \frac{\lvert CVE_{a_k}(t)\rvert}{\sum_{i=1}^{n} \lvert CVE_{a_i}(t)\rvert}$ | Relative frequency of vulnerabilities of the asset $a_k$ at a time $t$. From Eq. 4.6 | Ideally, the value of $M_4$ should be zero, or at least $M_4 \leq \frac{1}{n(t)}$, being $n(t)$ the number of assets in the SUT. This value can also be expressed as the percentage of vulnerabilities of asset $a_i$ respect to the total number of vulnerabilities in the SUT, $M_4\ (a_k, t) = \frac{\lvert CVE_{a_k}(t)\rvert}{\sum_{i=1}^{n} \lvert CVE_{a_i}(t)\rvert} \cdot 100$ |
| | $M_5(a_i, cwe_j, t) = \lvert CVE_{a_i \lvert cwe_j}(t)\rvert$ | Multiplicity of weakness $cwe_j$ of the asset $a_i$ at a time $t$. This metric represents the number of times a weakness is present among the vulnerabilities of the asset $a_i$. This is possible because a vulnerability can have associated the same weakness as other vulnerabilities. From Eq. 4.9 | Ideally, the value of $M_5$ should be zero, or at least, $M_5 \leq \frac{\lvert CVE_{A\lvert cwe_j}(t)\rvert}{n(t)}$, being $n(t)$ the number of assets in the SUT. The value of the metric could be further narrowed by assuming that $cwe_j$ will be present in all but one asset, so $M_5 \leq \frac{\lvert CVE_{A\lvert cwe_j}(t)\rvert}{n(t)-1}$ to be in acceptable values |
| | $M_6(A, cwe_j, t) = \lvert CVE_{A\lvert cwe_j}(t)\rvert$ | Multiplicity of weakness $cwe_j$ of the SUT $A$ at a time $t$. This metric represents the number of times a weakness is present among the vulnerabilities of the SUT $A$. From Eq. 4.11 | Ideally, the value of $M_6$ should be zero |
| WEAKNESSES | $M_7(A, t) = \lvert CWE_A(t)\rvert$ | Number of weaknesses in a SUT $A$ at time $t$. From Eq. 4.7 | Ideally, the value of $M_7$ should be zero (no weakness in $A$), but the lower the value of $M_7$, the better |
| | $M_8(A) = \sum_{t=1}^{T} \lvert CWE_A(t)\rvert = \sum_{t=1}^{T} M_7(A, t)$ | Number of weaknesses in a SUT $A$ throughout its entire life cycle $T$. This metric computes the accumulated value of weaknesses of a SUT throughout its entire life cycle. From Eq. 4.7 | The lower the value of $M_8$, the better |

#### 4.5.3.1 Automatic Inference of Root Causes

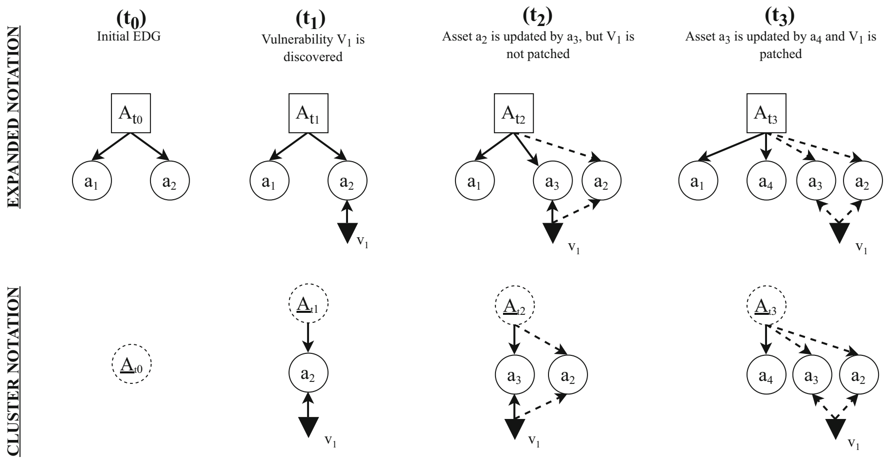
Each CWE natively contains information that is directly related to the root cause of a vulnerability. From this information, new requirements and test cases can be proposed.

#### 4.5.3.2 Spatial and Temporal Distribution of Vulnerabilities

The key feature of the EDG model is the addition of the temporal dimension in the analysis of vulnerabilities. This makes it possible to analyze the location of the vulnerabilities both in space (in which asset) and time (their recurrence), which allows us to track the state of the device throughout the whole life cycle. This approach also enables a further analysis of the SUT, by updating data in the model, such as new vulnerabilities that are found or new patches that are released.

Each time that a new vulnerability is found, or an asset is patched (i.e., via an update), the initial EDG is updated to reflect those changes. An example of this process can be seen in Fig. 4.6.

At time $t_0$, the initial graph of the SUT $A$ is depicted in Fig. 4.6. Because there is no vulnerability at that time, this graph can be simplified using the cluster notation, with just a cluster containing all assets. At time $t_1$, a new vulnerability that affects the asset $a_2$ is discovered. At time $t_2$, the asset $a_2$ is updated. This action creates a new version of asset $a_2$, asset $a_3$. Because the vulnerability was not corrected in the new update, both versions contain the vulnerability that was initially presented



EXPANDED NOTATION
CLUSTER NOTATION
(t0)
Initial EDG
(t1)
Vulnerability V1 is discovered
(t2)
Asset a2 is updated by a3, but V1 is not patched
(t3)
Asset a3 is updated by a4 and V1 is patched
At0
At1
At2
At3
a1
a2
a3
a4
V1

**Fig. 4.6** Representation of the temporal behavior in the graphical model using the two kinds of dependencies of the model. It is worth mentioning that these graphs could be further simplified by taking advantage of the cluster notation, as shown at the bottom of this figure

in asset $a_2$. Finally, at time $t_3$, the asset $a_3$ is updated to its new version $a_4$, and the vulnerability is corrected.

This approach enables a further analysis of the SUT, including updated data, according to new vulnerabilities that are found or new patches that are released.

#### 4.5.3.3 Patching Policies Prioritization Support

The EDG model provides a relative importance sorting of vulnerabilities by CVSS. Relying on the resulting value, it is possible to assist in the vulnerability patching prioritization process. Furthermore, the presence of an existing exploit for a known vulnerability can be also be taken into account, when deciding which vulnerabilities need to be patched first. A high CVSS value combined with an available exploit for a given vulnerability is a priority when patching.

## 4.6 Use Case

In this section, the EDG model and its metrics will be applied to perform a vulnerability assessment of the OpenPLC project, which will be the SUT.

OpenPLC is the first functional standardized open-source Programmable Logic Controller (PLC), both in software and hardware [72]. It was mainly created for research purposes in the areas of industrial and home automation, Internet of Things (IoT), and SCADA. Given that it is the only controller that provides its entire source code, it represents an engaging low-cost industrial solution – not only for academic research but also for real-world automation [2, 3].

### *4.6.1 Structure of OpenPLC*

The OpenPLC project consists of three parts:

1. **Runtime:** It is the software that plays the same role as the firmware in a traditional PLC. It executes the control program.
2. **Editor:** Application that is used to write and compile the control programs that will be later executed by the runtime.
3. **HMI Builder:** This software creates web-based animations that will reflect the state of the process, in the same manner as a traditional HMI.

When installed, the OpenPLC runtime executes a built-in webserver that allows OpenPLC to be configured and new programs for it to run to be uploaded.

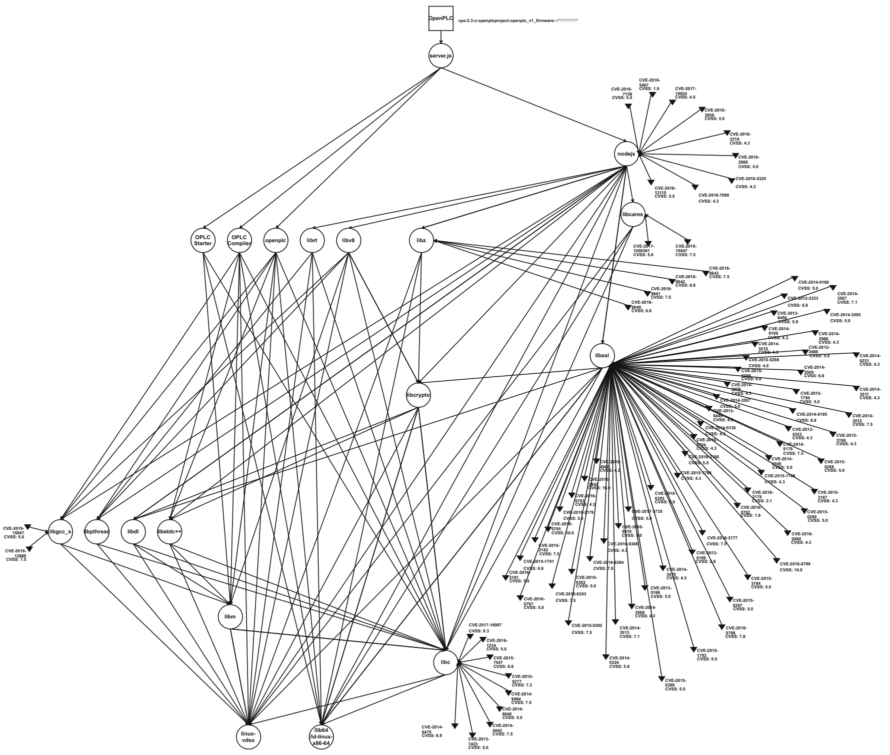


**Fig. 4.7** EDG for OpenPLC V1. Notice that, for simplicity, CWE and CAPEC values are omitted, and only the CPE identifier of the SUT is shown

### *4.6.2 Building the EDG*

For this use case, the setup consisted of OpenPLC installed on 14.04 LTS Ubuntu Linux in a virtual machine. All configuration options were by default.

Using the generated EDG for OpenPLC V1 shown in Fig. 4.7, we extracted the information about security updates (discarding updates that introduced more functionalities), for both `libssl` and `nodejs`. Table 4.3 shows the security updates and their date of availability for both `libssl` [39] and `nodejs` [40] for Ubuntu 14.04 LTS. There were two security updates available for the amd64 architecture for each asset.

From this data, we can extract that:

- Updates for `nodejs` were released before the updates for `libssl`.
- `libssl` shows more vulnerabilities than `nodejs`.
- The highest CVSS score in the period of this analysis is 5.

**Table 4.3** Update information of both `libssl` and `nodejs`

| Asset | 1st Update | Solved vulnerabilities (CVSS) | 2nd Update | Solved vulnerabilities (CVSS) |
|---|---|---|---|---|
| libssl | 2014/04/07 | CVE-2014-0076 (1.9)<br>CVE-2014-0160 (5.0) | 2018/12/06 | CVE-2018-5407 (1.9)<br>CVE-2018-0734 (4.3) |
| nodejs | 2014/03/27 | – | 2018/08/10 | CVE-2016-5325 (4.3) |

Then, the EDG for these two assets and their updates were built. Figure 4.8 shows the updates over time of the EDG, whereas Fig. 4.9 shows the final EDG with all the information included.

### *4.6.3 Analysis of the EDG*

Using Fig. 4.9, and Table 4.4, we can analyze the obtained EDG:

1. **Analysis of the induced EDG model:** The structure, assets, and dependencies are the focus of this first step.
   We can observe that `libssl` is used by `nodejs`, and they are not at the same level of the hierarchy. So vulnerabilities could propagate upward and downward through the EDG.
2. **Vulnerability analysis:** Vulnerability number, distribution, and severity are analyzed in this step. A proposal for vulnerability prioritization is also generated.
   We can highlight that `nodejs` had one vulnerability discovered after its first update, whereas `libssl` had vulnerabilities in both periods of time. We could argue that, as `nodejs` is the most accessible asset from the exterior, its vulnerabilities should be first addressed, even though the associated CVSS is not the highest one.
3. **Weaknesses analysis:** Finally, the root cause of each vulnerability is found. In this step, new requirements, test cases, and training activities are proposed based on the results of the analysis.
   Table 4.4 shows the root cause for each vulnerability. Using this data, new requirements (Table 4.5), test cases (Table 4.6) and training activities (Table 4.7) were proposed.

It is worth noticing that this use case is focused on reflecting the temporal evolution of the EDG. For this reason, metrics cannot be computed here, because of the low number of vulnerabilities available.

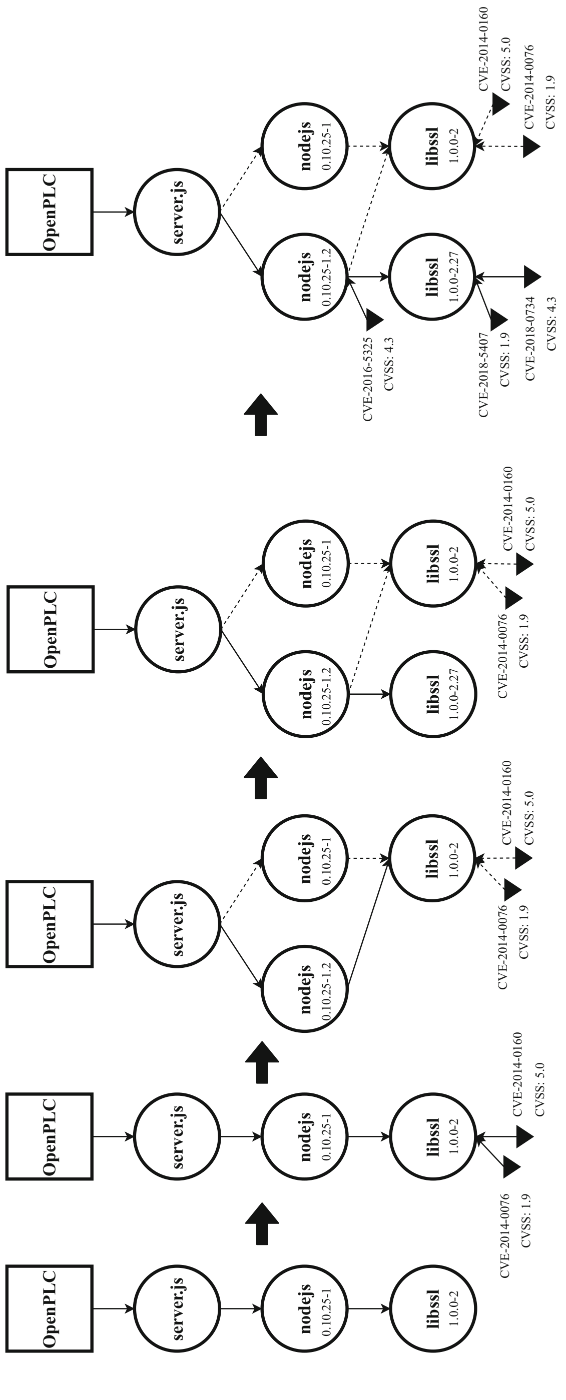


**Fig. 4.8** Temporal evolution of the EDG for OpenPLC V1 for both libss and nodejs

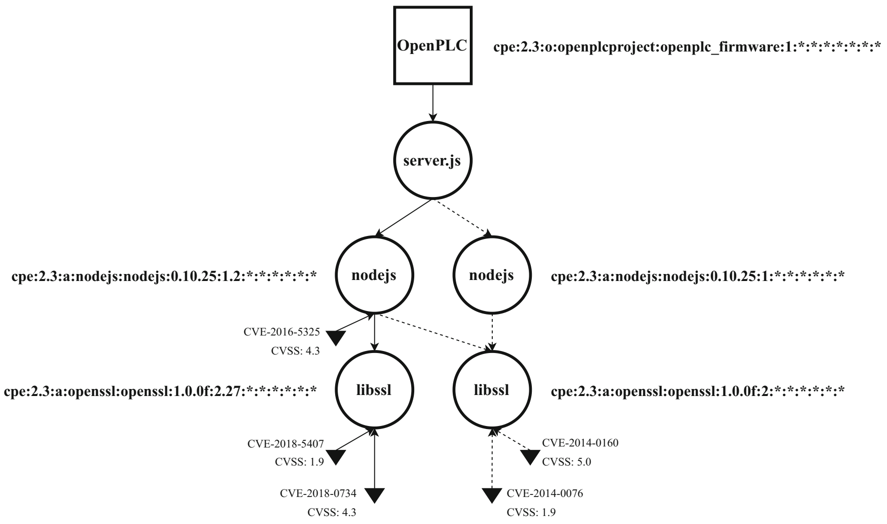

OpenPLC
cpe:2.3:o:openplcproject:openplc_firmware:1:*:*:*:*:*:*
server.js
nodejs
cpe:2.3:a:nodejs:nodejs:0.10.25:1.2:*:*:*:*:*:*
nodejs
cpe:2.3:a:nodejs:nodejs:0.10.25:1:*:*:*:*:*:*
CVE-2016-5325
CVSS: 4.3
libssl
cpe:2.3:a:openssl:openssl:1.0.0f:2.27:*:*:*:*:*:*
libssl
cpe:2.3:a:openssl:openssl:1.0.0f:2:*:*:*:*:*:*
CVE-2018-5407
CVSS: 1.9
CVE-2014-0160
CVSS: 5.0
CVE-2018-0734
CVSS: 4.3
CVE-2014-0076
CVSS: 1.9

**Fig. 4.9** Final EDG for libssl and nodejs integrating all the updates for Ubuntu Linux 14.04 for amd64 architecture

**Table 4.4** Relationship between vulnerabilities and weaknesses for both `libssl` and `nodejs`

| CVE | CVSS | CWE | Description |
|---|---|---|---|
| CVE-2014-0076 | 1.9 | CWE-310 | Cryptographic Issues |
| CVE-2014-0160 | 7.5 | CWE-119 | Improper restriction of operations within the Bounds of a Memory Buffer |
| CVE-2016-5325 | 6.1 | CWE-113 | Improper neutralization of CRLF Sequences in HTTP Headers (‘’HTTP Response Splitting”) |
| CVE-2018-0734 | 5.9 | CWE-327 | Use of a Broken or Risky Cryptographic algorithm |
| CVE-2018-5407 | 4.7 | CWE-203 | Observable discrepancy |
| | | CWE-200 | Exposure of sensitive information to an unauthorized actor |

**Table 4.5** An example of generated requirements for OpenPLC V1

| CWE ID | Requirements |
|---|---|
| CWE-119 | Use languages that perform their own memory management. |
| CWE-113 | Use an input validation framework. |
| CWE-113 | Assume all input is malicious. |
| CWE-119 | Replace unbounded copy functions with analogous functions that support length arguments. Create these if they are not available. |

## 4.7 Conclusions

The rapid evolution of industrial components, the paradigm of Industry 4.0, and the new connectivity features introduced by 5G technology increase the likelihood of cybersecurity incidents. These incidents have to be managed to limit or mitigate

**Table 4.6** Example of generated test cases for OpenPLC V1

| CAPEC ID | Test cases |
|---|---|
| CAPEC-119 | Check for buffer overflows through manipulation of environment variables |
| CAPEC-119 | Feed overly long input strings to the program to cause a buffer overflow, so the filter does not fail securely |
| CAPEC-119 | Create or manipulate a symbolic link file such that its contents result in out of bounds data. It could potentially overflow internal buffers with insufficient bounds checking |
| CAPEC-119 | Static analysis of the code: secure functions and buffer overflow |

**Table 4.7** Example of proposed training for OpenPLC V1

| CWE ID | Training |
|---|---|
| CWE-113, CWE-119 | Input validation strategies. |
| CWE-113, CWE-119 | Character encoding compatibility. |
| CWE-200 | Secure functions. |
| CWE-190 | Secure programming: memory management. |
| CWE-113, CWE-119 | System compartmentalization. |
| CWE-310 | Secure up-to-date cryptographic algorithms. |

their impact, and in most cases, they are a consequence of existing vulnerabilities. This scenario raises the need for a tool that enables a faster (tracking the vulnerability state over time) and more precise (detect root cause) response.

Vulnerability analysis is a critical task which ensures the security of industrial components. The EDG model that we reviewed performs continuous vulnerability assessment throughout the entire life cycle of industrial components. The model is built on (1) the directed graph representation of the internal structure of the device, (2) the set of quantitative metrics based on the Common Vulnerability Scoring System (CVSS), and (3) the algorithm to build the EDG for a given device. Metrics can be used by the model to improve the development process of the SUT, enhance its security, and track its status. The key feature of the EDG model is the addition of the temporal dimension in the analysis of vulnerabilities. The location of vulnerabilities can be analyzed in both space (in which asset) and time (their recurrence), which allows the state of the device to be tracked throughout the whole life cycle. The EDG model can be applied throughout the entire lifespan of a device to track vulnerabilities, identify new requirements, root causes, and test cases. It also helps prioritize patching activities.

## References

1. M. Alenezi, M. Zarour, On the relationship between software complexity and security. Int. J. Softw. Eng. Appl. **11**(1) (2020), https://aircconline.com/abstract/ijsea/v11n1/11120ijsea04.html

2. T. Alves, T. Morris, OpenPLC: an IEC 61,131–3 compliant open source industrial controller for cyber security research. Comput. Secur. **78**, 364–379 (2018). https://doi.org/https://doi.org/10.1016/j.cose.2018.07.007, https://www.sciencedirect.com/science/article/pii/S0167404818305388
3. T.R. Alves, M. Buratto, F.M. de Souza, T.V. Rodrigues, OpenPLC: an open source alternative to automation, in *IEEE Global Humanitarian Technology Conference (GHTC 2014)*, pp. 585–589 (2014). https://doi.org/10.1109/GHTC.2014.6970342, https://ieeexplore.ieee.org/document/6970342
4. M.A. Amutio, J. Candau, J.A. Mañas, MAGERIT V3.0. Methodology for Information Systems Risk Analysis and Management. Book I – The Method. National Standard, Ministry of Finance and Public Administration, Madrid, Spain (2014)
5. O. Andreeva, S. Gordeychik, G. Gritsai, O. Kochetova, E. Potseluevskaya, S. Sidorov, A. Timorin, Industrial control systems vulnerabilities statistics. Tech. rep., Kaspersky Lab (March 2016). https://doi.org/10.13140/RG.2.2.15858.66241
6. P. Arpaia, F. Bonavolontà, A. Cioffi, N. Moccaldi, Reproducibility enhancement by optimized power analysis attacks in vulnerability assessment of IOT transducers. IEEE Trans. Instrum. Meas. **70**, 1–8 (2021). https://doi.org/10.1109/TIM.2021.3107610, https://ieeexplore.ieee.org/document/9521880
7. A. Avizienis, J. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secure Comput. **1**(1), 11–33 (2004). https://doi.org/10.1109/TDSC.2004.2, https://ieeexplore.ieee.org/document/1335465
8. M. Ayaz, M. Ammad-Uddin, Z. Sharif, A. Mansour, E.H.M. Aggoune, Internet-of-things (IOT)-based smart agriculture: Toward making the fields talk. IEEE Access **7**, 129551–129583 (2019). https://doi.org/10.1109/ACCESS.2019.2932609, https://ieeexplore.ieee.org/document/8784034
9. N. Benias, A.P. Markopoulos, A review on the readiness level and cyber-security challenges in industry 4.0, in *2017 South Eastern European Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM)* (2017), pp. 1–5. https://doi.org/10.23919/SEEDA-CECNSM.2017.8088234, https://ieeexplore.ieee.org/document/8088234
10. B.A. Cheikes, D. Waltermire, K. Scarfone, NIST Interagency Report 7695 – Common Platform Enumeration: naming Specification Version 2.3. Nist interagency report, National Institute for Standards and Technology (NIST), Gaithersburg, Maryland (2011). https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=909010
11. CC: The Common Criteria for Information Technology Security Evaluation – Introduction and General Model. https://www.commoncriteriaportal.org/files/ccfiles/CCPART1V3.1R5.pdf
12. T.M. Chen, S. Abu-Nimeh, Lessons from Stuxnet. Computer **44**(4), 91–93 (2011). https://doi.org/10.1109/MC.2011.115, https://ieeexplore.ieee.org/document/5742014
13. K. Christidis, M. Devetsikiotis, Blockchains and smart contracts for the internet of things. IEEE Access **4**, 2292–2303 (2016). https://doi.org/10.1109/ACCESS.2016.2566339, https://ieeexplore.ieee.org/document/7467408
14. Common Criteria (CC): Part 3: Security Assurance Components. https://commoncriteriaportal.org/files/ccfiles/CCPART3V3.1R5.pdf
15. G. Culot, F. Fattori, M. Podrecca, M. Sartor, Addressing industry 4.0 cybersecurity challenges. IEEE Eng. Manag. Rev. **47**(3), 79–86 (2019). https://doi.org/10.1109/EMR.2019.2927559, https://ieeexplore.ieee.org/document/8758411
16. A. Dimitriadis, J.L. Flores, B. Kulvatunyou, N. Ivezic, I. Mavridis, Ares: automated risk estimation in smart sensor environments. Sensors **20**(16) (2020). https://doi.org/10.3390/s20164617, https://www.mdpi.com/1424-8220/20/16/4617
17. FIRST – global Forum of Incident Response and Security Teams: Common Vulnerability Scoring System (CVSS). https://www.first.org/cvss/
18. A. Fuller, Z. Fan, C. Day, C. Barlow, Digital twin: Enabling technologies, challenges and open research. IEEE Access **8**, 108952–108971 (2020). https://doi.org/10.1109/ACCESS.2020.2998358

19. I. Garitano, S. Fayyad, J. Noll, Multi-metrics approach for security, privacy and dependability in embedded systems. Wirel. Pers. Commun. (2015). https://doi.org/10.1007/s11277-015-2478-z, https://link.springer.com/article/10.1007%2Fs11277-015-2478-z
20. G. George, S.M. Thampi, A graph-based security framework for securing industrial IOT networks from vulnerability exploitations. IEEE Access **6**, 43586–43601 (2018). https://doi.org/10.1109/ACCESS.2018.2863244, https://ieeexplore.ieee.org/document/8430731
21. L. Gressl, C. Steger, U. Neffe, Design space exploration for secure IOT devices and cyber-physical systems. ACM Trans. Embed. Comput. Syst. **20**(4) (2021). https://doi.org/10.1145/3430372, https://doi.org/10.1145/3430372
22. M. Gupta, M. Abdelsalam, S. Khorsandroo, S. Mittal, Security and privacy in smart farming: challenges and opportunities. IEEE Access **8**, 34564–34584 (2020). https://doi.org/10.1109/ACCESS.2020.2975142, https://ieeexplore.ieee.org/document/9003290
23. V. Hassija, V. Chamola, V. Saxena, D. Jain, P. Goyal, B. Sikdar, A survey on IOT security: application areas, security threats, and solution architectures. IEEE Access **7**, 82721–82743 (2019). https://doi.org/10.1109/ACCESS.2019.2924045, https://ieeexplore.ieee.org/document/8742551
24. W. He, H. Li, J. Li, Unknown vulnerability risk assessment based on directed graph models: a survey. IEEE Access **7**, 168201–168225 (2019). https://doi.org/10.1109/ACCESS.2019.2954092, https://ieeexplore.ieee.org/abstract/document/8906081
25. J.I. Hejderup, A. Van Deursen, A. Mesbah, In Dependencies We Trust: How vulnerable are dependencies in software modules? Ph.D. thesis, Department of Software Technology, TU Delft (2015). http://resolver.tudelft.nl/uuid:3a15293b-16f6-4e9d-b6a2-f02cd52f1a9e
26. J. Homer, X. Ou, D. Schmidt, A sound and practical approach to quantifying security risk in enterprise networks. Tech. rep., Kansas State University (2009). https://www.cse.usf.edu/~xou/publications/tr_homer_0809.pdf
27. J. Homer, S. Zhang, X. Ou, D. Schmidt, Y. Du, S.R. Rajagopalan, A. Singhal, Aggregating vulnerability metrics in enterprise networks using attack graphs. J. Comput. Secur. **21**(4), 561–597 (2013). https://doi.org/10.3233/JCS-130475, https://content.iospress.com/articles/journal-of-computer-security/jcs475
28. J. Hu, S. Guo, X. Kuang, F. Meng, D. Hu, Z. Shi, I-HMM-based multidimensional network security risk assessment. IEEE Access **8**, 1431–1442 (2020). https://doi.org/10.1109/ACCESS.2019.2961997, https://ieeexplore.ieee.org/document/8941077
29. D. Hwang, P. Schaumont, K. Tiri, I. Verbauwhede, Securing embedded systems. IEEE Secur. Priv. **4**(2), 40–49 (2006). https://doi.org/10.1109/MSP.2006.51, https://ieeexplore.ieee.org/document/1621059
30. International Electrotechnical Commission: IEC 62443: Industrial Communication Networks–Network and System Security. Standard, IEC Central Office, Geneva, Switzerland (2010)
31. International Electrotechnical Commission: IEC 62443: Security for Industrial Automation and Control Systems – Part 4–1: Secure Product Development Lifecycle Requirements. Standard, International Electrotechnical Commission, Geneva, Switzerland (2018)
32. International Electrotechnical Commission: IEC 62443: Security for Industrial Automation and Control Systems – Part 4–2: Technical Security Requirements for IACS Components. Standard, International Electrotechnical Commission, Geneva, Switzerland (2019). https://www.isa.org/products/ansi-isa-62443-4-1-2018-security-for-industrial-au
33. ISO: ISO 8601:2019. Data and time – Representation for information interchange – Part 1: Basic rules. International Organization for Standardization, Geneva, Switzerland (2019). https://www.iso.org/standard/70907.html
34. ISO: ISO/IEC 13335-1:2004 – Information technology – Security techniques – Management of information and communications technology security – Part 1: Concepts and models for information and communications technology security management. International Organization for Standardization, Geneva, Switzerland (2019). https://www.iso.org/standard/70907.html
35. D. Kleidermacher, M. Kleidermacher, Practical methods for safe and secure software and systems development, in *Embedded Systems Security*, ed. by D. Kleidermacher, M. Kleidermacher (Newnes, Oxford, 2012). https://doi.org/https://doi.org/10.1016/B978-0-12-386886-2.00001-1, https://www.sciencedirect.com/science/article/pii/B9780123868862000011

36. R. Langner, Stuxnet: dissecting a cyberwarfare weapon. IEEE Secur. Priv. **9**(3), 49–51 (2011). https://doi.org/10.1109/MSP.2011.67
37. M. Lezzi, M. Lazoi, A. Corallo, Cybersecurity for industry 4.0 in the current literature: a reference framework. Comput. Ind. **103**, 97–110 (2018). https://doi.org/https://doi.org/10.1016/j.compind.2018.09.004, https://www.sciencedirect.com/science/article/pii/S0166361518303658
38. A. Longueira-Romero, R. Iglesias, J.L. Flores, I. Garitano, A novel model for vulnerability analysis through enhanced directed graphs and quantitative metrics. Sensors **22**(6) (2022). https://doi.org/10.3390/s22062126, https://www.mdpi.com/1424-8220/22/6/2126
39. C. Ltd., libssl1.0.0: Trusty (14.04): Ubuntu. https://launchpad.net/ubuntu/trusty/+package/libssl1.0.0/+index
40. C. Ltd., nodejs: Trusty (14.04): Ubuntu. https://launchpad.net/ubuntu/trusty/+package/nodejs/+index
41. M. Dekker, C. Karsberg, Guideline on Threats and Assets: Technical guidance on threats and assets in Article 13a. Tech. rep., European Union Agency For Network And Information Security (2015). https://www.enisa.europa.eu/publications/technical-guideline-on-threats-and-assets
42. P. Marwedel, Embedded systems foundations of cyber-physical systems, and the internet of things, in *Embedded System Design* (Springer Nature, Switzerland, 2018). https://doi.org/https://doi.org/10.1007/978-3-319-56045-8, https://link.springer.com/book/10.1007%2F978-3-319-56045-8
43. M.C. Parmelee, H. Booth, D. Waltermire, K. Scarfone, NIST Interagency Report 7696 – Common Platform Enumeration: Name Matching Specification Version 2.3. Nist interagency report, National Institute for Standards and Technology (NIST), Gaithersburg, Maryland (2011). https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=909008
44. A. Mathew, Network slicing in 5G and the security concerns, in *2020 Fourth International Conference on Computing Methodologies and Communication (ICCMC)*, pp. 75–78 (2020). https://ieeexplore.ieee.org/abstract/document/9076479
45. W. Matsuda, M. Fujimoto, T. Aoyama, T. Mitsunaga, Cyber security risk assessment on industry 4.0 using ICS testbed with AI and cloud, in *2019 IEEE Conference on Application, Information and Network Security (AINS)* (2019), pp. 54–59. https://doi.org/10.1109/AINS47559.2019.8968698, https://ieeexplore.ieee.org/document/8968698
46. S. McLaughlin, C. Konstantinou, X. Wang, L. Davi, A.R. Sadeghi, M. Maniatakos, R. Karri, The cybersecurity landscape in industrial control systems. Proc. IEEE **104**(5), 1039–1057 (2016). https://doi.org/10.1109/JPROC.2015.2512235, https://ieeexplore.ieee.org/document/7434576?reload=true&arnumber=7434576
47. N. Medeiros, N. Ivaki, P. Costa, M. Vieira, Software metrics as indicators of security vulnerabilities, in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)* (2017), pp. 216–227. https://doi.org/10.1109/ISSRE.2017.11, https://ieeexplore.ieee.org/document/8109088
48. MITRE Corporation, https://www.mitre.org/
49. MITRE Corporation: CAPEC – Common Attack Pattern Enumeration and Classification. https://capec.mitre.org/about/glossary.html
50. MITRE Corporation: CAPEC – Common Attack Pattern Enumeration and Classification. https://capec.mitre.org/
51. MITRE Corporation: CVE – Common Vulnerabilities and Exposures. https://cve.mitre.org/about/terminology.html
52. MITRE Corporation: CVE – Common Vulnerability and Exposures. https://cve.mitre.org/index.html
53. MITRE Corporation: CWE – Common Weakness Enumeration. https://cwe.mitre.org/about/faq.html
54. MITRE Corporation: CWE – Common Weakness Enumeration. https://cwe.mitre.org/index.html

55. S. Mumtaz, A. Alsohaily, Z. Pang, A. Rayes, K.F. Tsang, J. Rodriguez, Massive internet of things for industrial applications: addressing wireless IIOT connectivity challenges and ecosystem fragmentation. IEEE Ind. Electron. Mag. **11**(1), 28–33 (2017). https://doi.org/10.1109/MIE.2016.2618724, https://ieeexplore.ieee.org/document/7883984
56. L. MuÑoz-González, D. Sgandurra, M. Barrère, E.C. Lupu, Exact inference techniques for the analysis of bayesian attack graphs. IEEE Trans. Dependable Secure Comput. **16**(2), 231–244 (2019). https://doi.org/10.1109/TDSC.2016.2627033, https://ieeexplore.ieee.org/document/7885532
57. National Institute for Standards and Technology (NIST): CPE – Common Platform Enumeration. https://nvd.nist.gov/products/cpe
58. National Institute for Standards and Technology (NIST): National Vulnerability Database NVD – Vulnerabilities. https://nvd.nist.gov/vuln/full-listing
59. National Institute for Standards and Technology (NIST): vulnerability assessment – Glossary | CSRC. https://csrc.nist.gov/glossary/term/vulnerability_assessment
60. B.B. Nielsen, M.T. Torp, A. Møller, Modular call graph construction for security scanning of node.js applications, in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2021 (Association for Computing Machinery, New York, NY, USA, 2021), pp. 29–41. https://doi.org/10.1145/3460319.3464836, https://doi.org/10.1145/3460319.3464836
61. National Institute for Standards and Technology (NIST). https://www.nist.gov/
62. NIST – National Institute of Standards and Technology: National Vulnerability database (NVD). https://nvd.nist.gov/
63. M.O. Ojo, S. Giordano, G. Procissi, I.N. Seitanidis, A review of low-end, middle-end, and high-end IOT devices. IEEE Access **6**, 70528–70554 (2018). https://doi.org/10.1109/ACCESS.2018.2879615, https://ieeexplore.ieee.org/document/8528362
64. D. Papp, Z. Ma, L. Buttyan, Embedded systems security: threats, vulnerabilities, and attack taxonomy, in *2015 13th Annual Conference on Privacy, Security and Trust (PST)* (2015), pp. 145–152. https://doi.org/10.1109/PST.2015.7232966, https://ieeexplore.ieee.org/document/7232966
65. I. Pashchenko, H. Plate, S.E. Ponta, A. Sabetta, F. Massacci, Vulnerable open source dependencies: counting those that matter, in *Proceedings of the 12th International Symposium on Empirical Software Engineering and Measurement (ESEM)* (2018), https://dl.acm.org/doi/10.1145/3239235.3268920
66. S.E. Ponta, H. Plate, A. Sabetta, Detection, assessment and mitigation of vulnerabilities in open source dependencies. Empir. Softw. Eng. **25**(5), 3175–3215 (2020). https://doi.org/10.1007/s10664-020-09830-x, https://doi.org/10.1007/s10664-020-09830-x
67. N. Poolsappasit, R. Dewri, I. Ray, Dynamic security risk management using bayesian attack graphs. IEEE Trans. Dependable Secure Comput. **9**(1), 61–74 (2012). https://doi.org/10.1109/TDSC.2011.34, https://ieeexplore.ieee.org/document/5936075
68. O. Qingyu, L. Fang, H. Kai, High-security system primitive for embedded systems, in *2009 International Conference on Multimedia Information Networking and Security*, vol. 2 (2009), pp. 319–321. https://doi.org/10.1109/MINES.2009.48, https://ieeexplore.ieee.org/document/5368926
69. R.E. Sawilla, X. Ou, Identifying critical attack assets in dependency attack graphs, in Computer Security – ESORICS 2008, ed. by S. Jajodia, J. Lopez (Springer, Berlin/Heidelberg, 2008), pp. 18–34. https://link.springer.com/chapter/10.1007/978-3-540-88313-5_2#citeas
70. K. Shafique, B.A. Khawaja, F. Sabir, S. Qazi, M. Mustaqim, Internet of Things (IoT) for next-generation smart systems: a review of current challenges, future trends and prospects for emerging 5G-IoT scenarios. IEEE Access **8**, 23022–23040 (2020). https://doi.org/10.1109/ACCESS.2020.2970118, https://ieeexplore.ieee.org/document/9103025
71. L. Thames, D. Schaefer (eds.), Cybersecurity for Industry 4.0. Springer International Publishing (2017). https://doi.org/10.1007/978-3-319-50660-9, https://link.springer.com/book/10.1007/978-3-319-50660-9
72. Thiago Alves: OpenPLC Project. https://www.openplcproject.com/

73. A. Ustundag, E. Cevikcan, *Industry 4.0: Managing The Digital Transformation* (Springer International Publishing, 2018). https://doi.org/10.1007%2F978-3-319-57870-5
74. J. Viega, H. Thompson, The state of embedded-device security (spoiler alert: It's bad). IEEE Secur. Priv. **10**(5), 68–70 (2012). https://doi.org/10.1109/MSP.2012.134, https://ieeexplore.ieee.org/document/6322974?section=abstract
75. Y. Xin, L. Kong, Z. Liu, Y. Chen, Y. Li, H. Zhu, M. Gao, H. Hou, C. Wang, Machine learning and deep learning methods for cybersecurity. IEEE Access **6**, 35365–35381 (2018). https://doi.org/10.1109/ACCESS.2018.2836950, https://ieeexplore.ieee.org/document/8359287
76. S. Zhang, X. Ou, A. Singhal, J. Homer, An empirical study of a vulnerability metric aggregation method. Tech. rep., Kansas State Univ Manhattan (2011). https://www.cse.usf.edu/~xou/publications/stmacip11.pdf
77. I. Zografopoulos, J. Ospina, X. Liu, C. Konstantinou, Cyber-physical energy systems security: threat modeling, risk assessment, resources, metrics, and case studies. IEEE Access **9**, 29775–29818 (2021). https://doi.org/10.1109/ACCESS.2021.3058403, https://ieeexplore.ieee.org/document/9351954

# Chapter 5
# Metamorphic Testing for Verification and Fault Localization in Industrial Control Systems

**Gaadha Sudheerbabu, Tanwir Ahmad, Dragos Truscan, and Jüri Vain**

**Abstract** Security verification of software systems is vital to ensure they are resilient against targeted attacks. Any vulnerability in the software should be discovered, classified, and resolved promptly to ensure the operational correctness and functional safety of the system. However, testing and program debugging of complex industrial control systems are often challenging due to the test oracle problem. In this work, we discuss an integrated method for test generation and fault localization using metamorphic testing. Our method extracts metamorphic relation from the system specification and uses it as the derived test oracle to distinguish the successful and failed tests for spectrum-based fault localization. The proposed approach consists of two phases: a test generation phase using metamorphic testing and a fault localization phase to assist with the root cause analysis and failure diagnosis. The method is exemplified on a load position system without explicit specifications of the test oracle, and the results show that it is effective in discovering vulnerabilities in the application and significantly assists the developers with root cause analysis of identified faults that reduces the overall failure diagnosis effort.

**Keywords** Metamorphic testing · Spectrum-based fault localization · Safety and Security testing

## 5.1 Introduction

Industrial control systems (ICSs) operating in safety and security-critical applications are at serious risk of cyberattacks due to the expansion of attack surface when increasing automation levels are implemented for operational efficiency. A

G. Sudheerbabu · T. Ahmad · D. Truscan (✉)
Åbo Akademi University, Turku, Finland
e-mail: gaadha.sudheerbabu@abo.fi; tanwir.ahmad@abo.fi; dragos.truscan@abo.fi

J. Vain
Tallinn University of Technology, Tallinn, Estonia
e-mail: juri.vain@ttu.ee

A. Sadovykh et al. (eds.), *CyberSecurity in a DevOps Environment*,
https://doi.org/10.1007/978-3-031-42212-6_5

software system's vulnerabilities can be identified in its configuration, implemented code, or overall design. If exploited, vulnerabilities in such software systems can compromise their confidentiality, integrity, and availability or result in software failures leading to substantial financial losses. To ensure software systems are resilient against targeted attacks, any vulnerabilities in the software should be discovered, classified, and resolved promptly as early as possible in the stages of the software development life cycle to mitigate the risk of passing them to possible exploitation materializing in real loss.

Programmable logic controllers (PLC) are employed in complex industrial control systems, such as heavy machinery equipment control, nuclear power plants, energy distribution networks, rail automation, etc. There have been incidents reported about cyberattacks on nuclear power plants (NPP), such as the emergency shutdown of the Brown Ferry NPP in 2006, Hatch NPP in 2008, and the Stuxnet worm attack on the Natanz nuclear facility in 2010, which emphasizes the importance of security testing of industrial control systems.

Typically, the safety and security aspects of ICSs are addressed differently, and several definitions and distinctions of the two concepts have been attempted in literature. Maybe the most notable is [1], which distinguishes between the two as follows: *"Security is concerned with the risks originating from the environment and potentially impacting the system, whereas safety deals with the risks arising from the system and potentially impacting the environment."* and *"Security typically addresses malicious risks while safety addresses purely accidental risks."* However, the border between the two is rather fuzzy since, in many cases, existing safety issues can be triggered via security attacks [2].

For instance, an investigative study by Lim, Bernard, et al. [3] demonstrated the impacts of a cyberattack on a Tricon PLC system of a nuclear power plant. Their research revealed possible ways to trigger an attack and exploit the Tricon PLC vulnerabilities that use a Triple Modular Redundant (TMR) architecture. The findings from the study show that using the types of attack: (i) latent failure attack and (ii) immediate failure attack, the control logic of the Tricon system can be altered, resulting in a common-mode failure. In a different study [4], the authors also pinpoint that the security of the PLC system can be affected by control logic modification, either via program code modification or program input manipulation. Such code-level vulnerabilities can be introduced by different weaknesses, for instance, race conditions, uninitialized, hard-coded, or unused variables, improper input validation, etc. [4, 5] and, in the end, can affect security goals such as confidentiality, integrity, and availability.

These examples make evident that in order to prevent future cyberattacks on PLC systems, developing strategies for verifying and validating their functional safety assurance depending on security aspects must be seamlessly integrated into their development and maintenance processes. In fact, the ISA/IEC62443-4 standard for security for industrial automation and control systems [6] explicitly specifies that in order to ensure system integrity, each component should validate its input that directly impacts the action of that component. Also, the safety standard IEC 61508 [7] includes requirements to address cyber security in safety instrumented systems

(SIS) to be examined during the process hazards and risk assessment. Any hazardous events that have significant consequences on plant operation disruption or damage, personnel injury, or fatality are considered safety and security issues.

In many cases, security weaknesses are root caused in the code-level faults, especially caused by bad programming practices or code issues [8]. Once a vulnerability is identified, it is of utmost importance to localize its root cause and fix it. To this extent, fault localization is known to be a time-consuming and tedious process [9], and several approaches have been proposed to address this issue [10]; however, only a few in the area of ICSs based on PLCs.

In order to address the above aspects, we extend our previous approach of using metamorphic testing for ICSs [11] with a fault localization approach to allow us to pinpoint the root cause of the failing tests. The overall contributions of the approach consist in using metamorphic testing for PLC-based systems without an explicit oracle, combining metamorphic testing and fault localization techniques for PLC-code, and tool support for the approach. Although in this chapter we will exemplify our approach to a PLC system, the approach in itself is generic, and it can be extended and applied to a wide spectrum of software systems.

The chapter will briefly introduce metamorphic testing and fault localization techniques in Sect. 5.2, and then it will present our approach in Sect. 5.3. We exemplify and evaluate our approach in Sect. 5.4 on an industrial system that determines the position of a hanging load attached to the hoisting frame of a crane. We discuss the threats to validity in Sect. 5.5 and related work in Sect. 5.6, and we draw conclusions in Sect. 5.7.

## 5.2 Prerequisites

This section describes the metamorphic testing technique and the approaches of spectrum-based fault localization (SBFL) and program slicing.

### *5.2.1 Metamorphic Testing*

Metamorphic testing (MT) was introduced by Chen et al. [12] as a solution to test systems without explicit specification of the test oracle. In MT, the behavioral or functional properties of the system are defined by posing a hypothesis about using generic relations known as *metamorphic relations (MRs)* between different sets of inputs and their expected outputs. An MR is composed of two parts: an *input relation* and *output relation* [13]. An input relation represents the relation between the inputs of the source and follow-up test cases, whereas an output relation represents the relation between the expected outputs of the source and follow-up test cases. A *source test case* is the first set of tests performed using *seed inputs*. The seed inputs are transformed into *morphed inputs*. The *follow-up test cases* are performed using these *morphed inputs*. In addition, an *implication* between the outputs of source and follow-up test cases is needed to specify the impact of input transformations on their corresponding outputs.

A relatively recent study [14] shows that metamorphic testing becomes a sound alternative for testing systems without explicit oracles and that it has been applied successfully to many application domains, including embedded systems, web applications, computer graphics, and simulation and modeling.

## 5.2.2 *Fault Localization*

Fault localization is the process of identifying the potential fault-triggering program elements, and it can assist the developer by reducing the effort in the root cause analysis and program repair. Different fault localization techniques have been presented in a relatively recent survey [10]. Of those, in this work, we are going to use spectrum-based fault localization and program slicing, as discussed below.

### 5.2.2.1 Spectrum-Based Fault Localization

(SBFL) [15] is a popular fault localization technique that examines the execution of the program under test by collecting run-time measurements and uses them for program debugging and repair. The method relies on comparing the difference between the execution path of the program between successful and failed tests and examining the path taken by failed tests to identify potential faulty locations in the program. This information collected during program execution, known as program spectrum [16], is then used to calculate a suspiciousness metric that pinpoints the suspicious fault-triggering parts of the program under test. Different types of program spectra information used for SBFL are described in [17] and summarized in Table 5.1.

Recent surveys [18, 19], on SBFL discuss several suspiciousness metrics such as Tarantula, Ample, D*2, GP, OP1, OP2, SBI, Jaccard, Ochiai, and Wong 1–3 are regarded as the most effective in pinpointing fault locations.

### 5.2.2.2 Program Slicing

Program slicing [20, 21] is a fault localization technique that focuses on analyzing the relevant part of the program referred to as a *slice* that may contain a fault. The four primary categories of slicing techniques are static, dynamic, execution, and conditioned slicing [22–24]. Dynamic slicing and execution slicing have been widely applied for program debugging. Xie, Xiaoyuan, et al. [25] proposed the concept of a metamorphic slice by integrating metamorphic testing with dynamic and

**Table 5.1** Types of program spectra

| Mnemonic | Name | Description |
|---|---|---|
| BHS | Branch hit spectra | Conditional branches that were executed |
| BCS | Branch count spectra | Number of times each conditional branches was executed |
| CPS | Complete path spectra | Complete path that was executed |
| PHS | Path hit spectra | Number of times each intraprocedural, loop-free path was executed |
| PCS | Path count spectra | Number of times each intraprocedural, loop-free path was executed |
| DHS | Data-dependence hit spectra | Definition-use pairs that were executed |
| DCS | Data-dependence count spectra | Number of times each definition-use, pair was executed |
| ETS | Execution trace spectra | Execution trace that were produced |
| OPS | Output spectra | Output that were produced |

execution slicing in combination with SBFL. They defined dynamic metamorphic slice, *d_mslice* and execution metamorphic slice, *e_mslice* as follows:

- Given a variable $v$, the *dynamic mslice*, $d\_mslice(v, MR, T^S)$ is the union of all $d\_slice(v, t)$, where $d\_slice$ is the set of statements of a program which affect the value of the variable $v$, $t$ is a test case of the metamorphic test group, $T^S = \{t_1^S, t_2^S, ..., t_{ks}^S\}$ and $T^F = \{t_1^F, t_2^F, ..., t_{kf}^F\}$ are the set of source test cases and follow-up test cases for its metamorphic relation $MR$.

$$d\_mslice(v, MR, T^S) = \left(\bigcup_{i=1}^{ks} d\_slice(v, t_i^S)\right) \cup \left(\bigcup_{i=1}^{kf} d\_slice(v, t_i^F)\right)$$

- The *execution mslice*, $e\_mslice(MR, T^S)$, is the union of all $e\_slice(t)$, where $t$ is a test case of the metamorphic test group:

$$e\_mslice(MR, T^S) = \left(\bigcup_{i=1}^{ks} e\_slice(t_i^S)\right) \cup \left(\bigcup_{i=1}^{kf} e\_slice(t_i^F)\right)$$

Metamorphic testing using a defined MR ensures that a violation or non-violation of MR is available as a test verdict for each *d_mslice* or *e_mslice* in a metamorphic test suite.

## 5.3 Overview of the Approach

We propose an approach for the verification of ICSs by using metamorphic tests and vulnerability localization by integrating spectrum-based fault location with metamorphic slicing that can be integrated into the development and operations phase of ICSs. The integration of the proposed approach in the DevOps life cycle is presented in Fig. 5.1.

Our combined approach consists of two phases: a metamorphic testing phase and a fault localization phase. The *metamorphic testing phase* comprises a test generator that can use seed inputs either designed manually or extracted from logs at the monitoring stage of the operations cycle in order to generate metamorphic test inputs. It also extracts the metamorphic relations from the software requirements specification (SRS) to determine the verdict of the metamorphic tests. In the *fault localization phase*, a set of passed and failed tests from the first phase and the source code of the system under test are given as input to a fault localizer to identify the location of the fault(s). The findings from the localizer assist the developers in debugging and feedback-based program repair. The first phase follows a black box approach, whereas the second follows a white box approach.

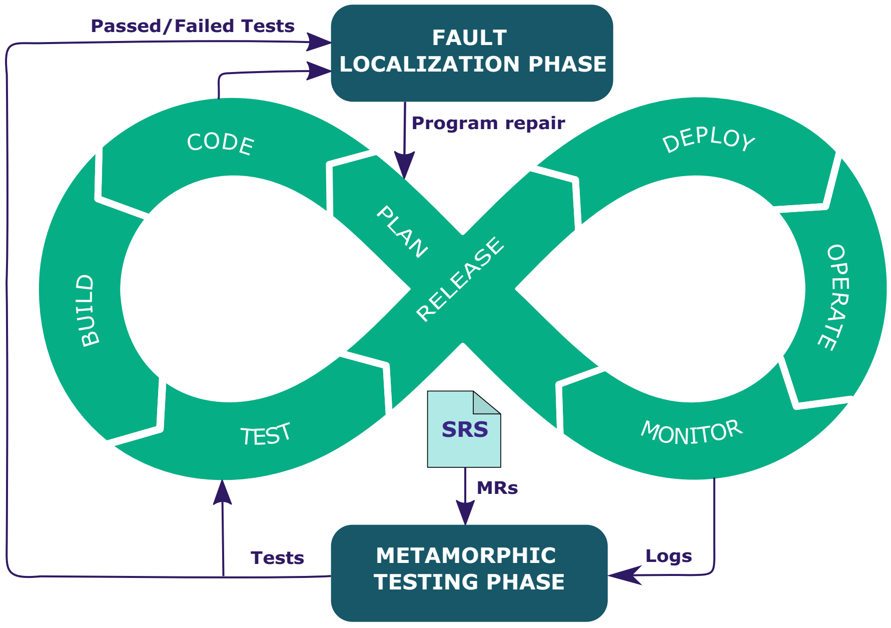


**Fig. 5.1** Verification and vulnerability localization in DevOps

### 5.3.1 Metamorphic Testing Phase

As the first contribution of this work, we extend the definition of metamorphic relation by Chen et al. [26]. We define a metamorphic relation $R$ as being composed of two relations, $R_{in}$ and $R_{out}$, on the inputs and, respectively, the outputs of the system under test. The satisfiability of MR output relation $R_{out}$ by outputs $Y_i$ and $Y_j$ also presumes that their corresponding seed $X_i$ and morphed inputs $X_j$ satisfy respectively MR input relation $R_{in}$. That is, given $\forall(x_i, x_j)$, $f(x_i) = y_i$ and $f(x_j) = y_j$, then $R_{in}(x_i, x_j) \Rightarrow R_{out}(y_i, y_j)$, where $f$ denotes the function that creates outputs $(y_i, y_j)$ in response to inputs $(x_i, x_j)$ and $R_{in}$ is input MR and $R_{out}$ is output MR.

Concretely, given two sets of inputs $X_s^C, X_m^C \in X^n$ that satisfy a given constraint $C$ on the input space and which are satisfying an input relation $R_{in}(X_s^C, X_m^C)$, an output relation $R_{out}$ should hold for any corresponding output of the system, that is, $R_{out}(Y_s^C, Y_m^C)$, where $Y_s^C = f(X_s^C)$ and $Y_m^C = f(X_m^C)$. Furthermore, we consider $R_{out}$ to be of any of the types defined in [27]: equivalence, equality, subset, disjoint, complete, and difference.

As a running example, we use a *multiply(x,y,z)* program, which calculates the product of three integer numbers passed as the input parameters. We extract the metamorphic relation from the associative and commutative properties of multiplication, meaning that any permutation of the input parameters should yield the same result as the original combination. Therefore, by applying the previous definitions, the input relation can be formulated as

$R_{in} = \{(X1, X2) | Permute(X1) = X2\}$ whereas the metamorphic relation is as follows:

$R = \{(X1, X2, Y1, Y2) | Permute(X1) = X2 \Rightarrow Y1 = Y2\}$, where the output relation $R_{out}$ is equality.

As a second contribution, we define our MT approach to consist of two steps, as shown in Fig. 5.2. In the *exploration step*, seed and, respectively, morphed inputs $X_s^C, X_m^C$ are created from X satisfying constraints $C_s, C_m$ respectively, which specify $R_{in}$ and are specific to the system under test (SUT). Then $X_s^C, X_m^C$ are executed against the SUT and the corresponding seed output $Y_s^C$ and respectively morphed output $Y_m^C$ are collected, and the satisfiability of $R_{out}(Y_s^C, Y_m^C)$ is checked, where $Y_s^C = f(X_s^C)$ and $Y_m^C = f(X_m^C)$.

For instance, in the case of the multiply program, if we consider (1,2,3) as seed input, we create (1,3,2) (2,1,3), (2,3,1), (3,1,2) and (3,2,1) as morphed input. Table 5.2 shows a few samples of seed-input, morphed-input, and the test inputs that failed or satisfied the MR for a seed input consisting of two tests: (2,3,4) and (−2,3,−2).

From those pairs of seed and morphed inputs $(X_{s_i}^C, X_{m_i}^C)$ which fail the initial MR, we manually extract, in the exploitation step, **fault-inducing inputs** of the input space. Based on them, we define $C'_m$ as a more restrictive constraint that encodes the refinement of $R_{in}$ to be satisfied by morphed inputs $X_m^{C'}$ which we use to verify the output metamorphic relation $R_{out}(Y_s^{C'}, Y_m^{C'})$, where $Y_s^{C'} = f(X_s^{C'})$

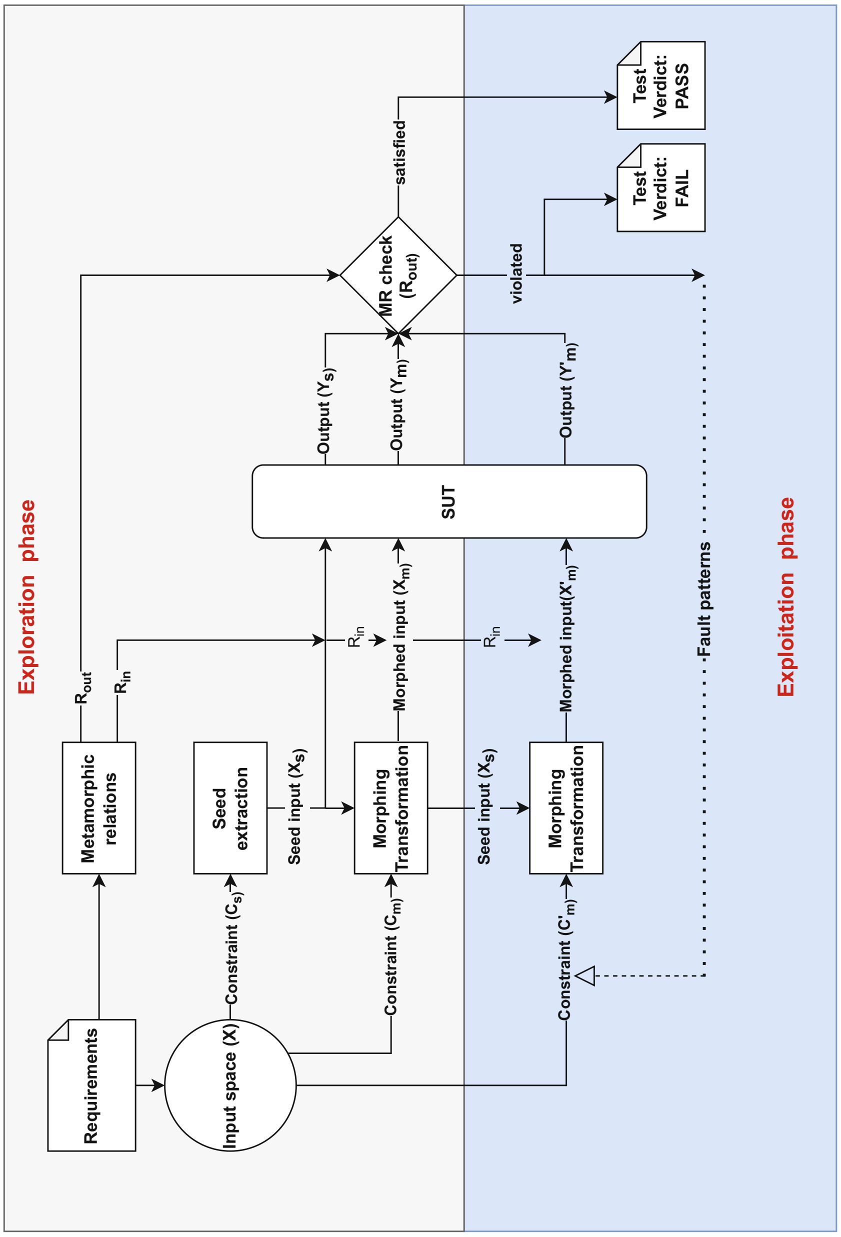


**Fig. 5.2** Overview of the metamorphic testing phase

**Table 5.2** Test results of program *multiply(x, y, z)* with metamorphic test inputs

| Seed input | Morphed input | MR status |
|---|---|---|
| (2, 3, 4) | (2, 4, 3) | Pass |
| | (3, 2, 4) | Pass |
| | (3, 4, 2) | Pass |
| | (4, 2, 3) | Pass |
| | (4, 3, 2) | Pass |
| (−2, 3, 2) | (2, 3, −2) | **Fail** |
| | (2, −2, 3) | **Fail** |
| | (3, 2, −2) | **Fail** |
| | (3, −2, 2) | **Fail** |
| | (−2, 2, 3) | Pass |

and $Y_m^{C'} = f(X_m^{C'})$. For the *multiply* program, we notice that tests with a negative value in the second or third input parameter violate the metamorphic relation during the exploration phase. Therefore, we generate more morphed test inputs in the exploitation phase to check the satisfiability of this constraint.

To recap, the novelty of this phase stands in the fact that $C'_m$ allows us to define a refined morphed input that tests the system with more precision and effectiveness by focusing the testing on the parts of the input with a higher probability of discovering faults as will be demonstrated in Sect. 5.4.

### 5.3.2 *Fault Localization Phase*

The fault localization phase in our work is a white box approach, which takes as input the source code of the component under test and two test suites extracted from the metamorphic testing phase: a test suite with passed tests and a test suite with failed tests. These test suites are executed against an instrumented version of the program and are used to build program spectra information based on which the suspiciousness score is calculated. The suspiciousness scores allow one to identify which parts of the code are more frequently executed by failed tests. The code statements with the highest suspiciousness score are extracted and used for data-flow analysis, and a fault report is generated. The main steps in the fault localization phase are depicted in Fig. 5.3 and explained as follows.

#### 5.3.2.1 Test Selection

In this step, a portion of refined morphed inputs from the set of tests in the metamorphic testing phase is used. These tests are classified as passed and failed based on the *violation* or *non-violation of MR*, defined as the test oracle. In the current approach, we select an equal number of passed and failed tests from the two

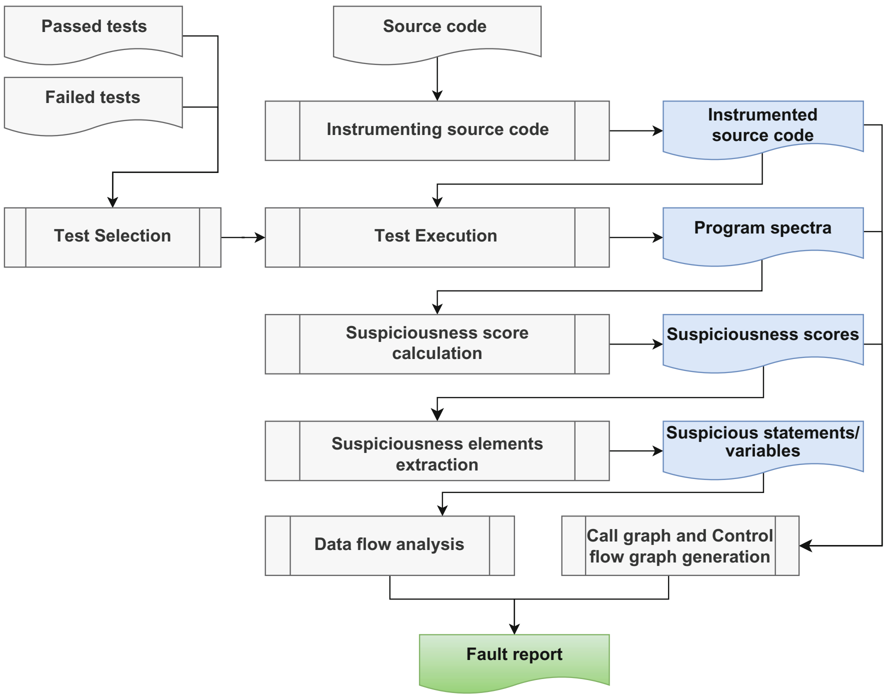

Passed tests
Failed tests
Source code
Instrumenting source code
Instrumented source code
Test Selection
Test Execution
Program spectra
Suspiciousness score calculation
Suspiciousness scores
Suspiciousness elements extraction
Suspicious statements/ variables
Data flow analysis
Call graph and Control flow graph generation
Fault report

**Fig. 5.3** Overview of the fault localization phase

test sets since, based on previous empirical results, it will make the fault analysis more clear and give equal fairness to each test set.

To exemplify, we choose from Table 5.2 four passed tests (2, 3, 4), (2, 4, 3), (3, 2, 4), and (3, 4, 2) and four failed tests (3, −2, 2), (3, 2, −2), (2, −2, 3), and (2, 3, −2).

### 5.3.2.2 Instrumenting Source Code

The source code of the component under test, including all its function blocks and functions, is instrumented by adding counter variables to each control statement in the code (as shown in Fig. 5.4).

The *flag* Boolean variable is used to control the update of the counters between successive inputs values globally and to avoid the increment of the counters during PLC update cycles. Whenever these variables are executed during the test execution, they will be incremented. The counter variable added in each program is an array of length equal to the number of branches in the program. The element index of the counter array is referred to as *Block ID*. A *Block ID* represents the basic block of statements associated with the corresponding counter element and branch

**ST code** → **Instrumented ST code**

```
FUNCTION multiply : REAL

VAR_INPUT
    in_var1        : REAL;
    in_var2        : REAL;
    in_var3        : REAL;
END_VAR

VAR_OUTPUT
    out_product   : REAL;
END_VAR

VAR
    x              : REAL;
    y              : REAL;
    z              : REAL;
    rxy            : REAL  := 0.0;
    rxyz           : REAL  := 0.0;
END_VAR

1:   x := in_var1;
2:   y := in_var2;
3:   z := in_var3;
4:   FOR i:=1 TO ABS(x) DO
5:     IF x >= 0 THEN
6:       rxy := rxy + y;
7:    ELSE
8:       rxy := -(rxy + y);
9:    END_IF
10: END_FOR
11: FOR j:=1 TO ABS(z) DO
12:    IF z >= 0 THEN;
13:       rxyz := rxyz + rxy;
14:    ELSE
15:       rxyz := rxyz + rxy;
16:    END_IF
17: END_FOR
18: out_product := rxyz;
```

```
FUNCTION multiply : REAL

VAR_INPUT
    in_var1        : REAL;
    in_var2        : REAL;
    in_var3        : REAL;
END_VAR

VAR_OUTPUT
    out_product   : REAL;
END_VAR

VAR
    x              : REAL;
    y              : REAL;
    z              : REAL;
    rxy            : REAL  := 0.0;
    rxyz           : REAL  := 0.0;
END_VAR

1:   x := in_var1;
2:   y := in_var2;
3:   z := in_var3;
4:   FOR i:=1 TO ABS(x) DO
5:     IF (flag)  THEN c[1] := c[1] + 1; END_IF
6:     IF x >= 0 THEN
7:        IF (flag)  THEN c[2] := c[2] + 1; END_IF
8:       rxy := rxy + y;
9:    ELSE
10:       IF (flag)  THEN c[3] := c[3] + 1; END_IF
11:       rxy := -(rxy + y);
12:    END_IF
13:  END_FOR
14:  FOR j:=1 TO ABS(z) DO
15:     IF (flag)  THEN c[4] := c[4] + 1; END_IF
16:     IF z >= 0 THEN;
17:        IF (flag)  THEN c[5] := c[5] + 1; END_IF
18:        rxyz := rxyz + rxy;
19:     ELSE
20:        IF (flag)  THEN c[6] := c[6] + 1; END_IF
21:        rxyz := rxyz + rxy;
22:     END_IF
23:  END_FOR
24:  out_product := rxyz;
```

**Fig. 5.4** Instrumentation of the structured text code for program `multiply(x, y, z)`

condition. The application declares the counter variables at the program level as global variables. This approach provides a run-time measurement of the number of times a basic block is executed during a test session for a specific metamorphic group of test inputs.

#### 5.3.2.3 Test Execution

These test suites are then executed separately against the SUT. The counter values obtained from each test session are used to identify the difference between execution paths and to calculate the suspiciousness scores.

#### 5.3.2.4 Suspiciousness Scores Calculation

Value of the counters collected during the test execution is used to calculate the program spectra and the suspiciousness score at the basic block level, such as branch hit spectra (BHS) and branch count spectra (BCS). The rationale behind this is that if the execution of a program element tends to be more frequent in failed tests, the more likely it is to be faulty and, consequently, higher the suspiciousness score.

In this work, we define a new suspiciousness metric, $s_{avg}$, with a value between 0 and 1, defined as the arithmetic mean of the three maximal metrics Tarantula, Ochiai, and Jaccard metrics:

$$s_{avg} = (s_{Ochiai} + s_{Jaccard} + s_{Tarantula})/3 \tag{5.1}$$

These three metrics are most well-known for their fault localization effectiveness and are widely used in empirical studies [25, 28]. The formulas for the suspiciousness metrics used are listed in Table 5.3.

The suspiciousness metric formula is based on four variables [18] that are defined as follows:

- **ef**: the number of times a statement is executed (e) in failed tests
- **ep**: the number of times a statement is executed (e) in passed tests
- **nf**: the number of times a statement is not executed (n) in failed tests
- **np**: the number of times a statement is not executed (n) in passed tests

#### 5.3.2.5 Suspicious Elements Extraction

In this step, the suspicious basic blocks in each program are assigned a score based on the suspiciousness metric formula used. Following this, the *execution mslice* and *dynamic mslice* at the program level are extracted for further analysis.

**Table 5.3** Suspiciousness metric and their formulas

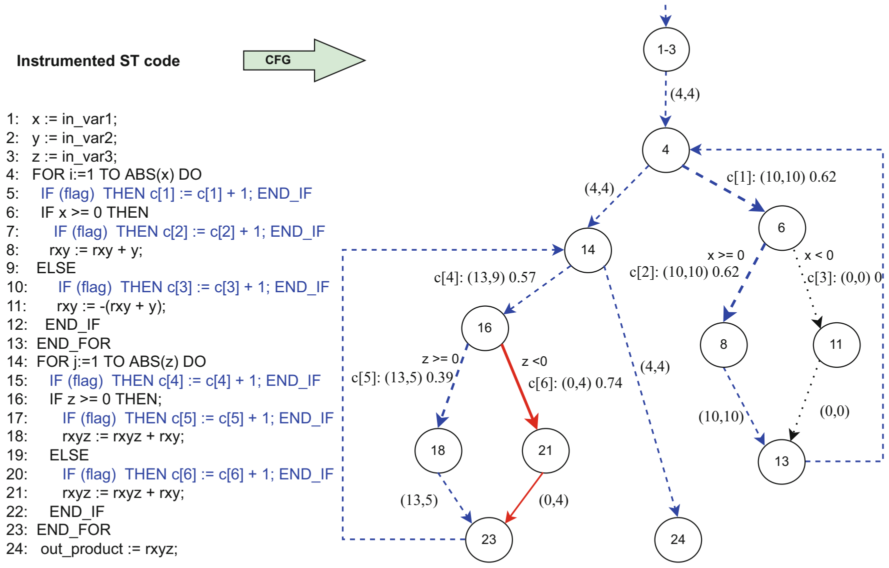
| Suspiciousness metric | Formula |
|---|---|
| Ochiai | $\frac{ef}{\sqrt{(ef+nf)\cdot(ef+ep)}}$ |
| Jaccard | $\frac{ef}{(ef+ep+nf)}$ |
| Tarantula | $\frac{\frac{ef}{(ef+nf)}}{\frac{ef}{(ef+nf)}+\frac{ep}{(ep+np)}}$ |

For the *multiply* program (Fig. 5.4), the *execution mslice* (refer Sect. 5.2.2.2) and *dynamic mslice* (refer Sect. 5.2.2.2) are extracted from the instrumented source code as follows: an execution mslice is set of union of statements in source and follow-up test execution, that is, $\{s_1, s_2, s_3, s_4, s_6, s_8, s_{13}, s_{14}, s_{16}, s_{18}, s_{21}, s_{23}, s_{24}\}$, where $s_i$ is a statement of the program in (Fig. 5.4). Considering the *execution mslice* of multiply program, the statement $s_{21}$ is the one associated with the highest suspicious score, and variable rxyz propagates the error to the output. A dynamic mslice of the variable `rxyz` from a test case $t$ belonging to failed tests group is the set of statements that affected it, that is, $\{s_{21}, s_{24}\}$.

#### 5.3.2.6 Call Graph and Control-Flow Graph Generation

From the test execution, we collect both the number of times each function block and function is called (as shown in Fig. 5.5) and calculate the suspiciousness score for each function block and function. The suspiciousness score of a function block/function is calculated as the highest suspiciousness score of all the basic blocks in it. A call graph of the component under test and an annotated control-flow graph (CFG) enable the developers to perform static and dynamic analysis in program debugging. The graphs are combined with the suspiciousness scores and provided to the domain expert for further analysis.

The call graph indicates which programs are executed by the PLC main program, their call order, and the call location from which the inter-procedural calls are



```
1:  x := in_var1;
2:  y := in_var2;
3:  z := in_var3;
4:  FOR i:=1 TO ABS(x) DO
5:    IF (flag)  THEN c[1] := c[1] + 1; END_IF
6:    IF x >= 0 THEN
7:       IF (flag)  THEN c[2] := c[2] + 1; END_IF
8:       rxy := rxy + y;
9:   ELSE
10:      IF (flag)  THEN c[3] := c[3] + 1; END_IF
11:      rxy := -(rxy + y);
12:   END_IF
13:  END_FOR
14:  FOR j:=1 TO ABS(z) DO
15:    IF (flag)  THEN c[4] := c[4] + 1; END_IF
16:    IF z >= 0 THEN;
17:       IF (flag)  THEN c[5] := c[5] + 1; END_IF
18:       rxyz := rxyz + rxy;
19:    ELSE
20:       IF (flag)  THEN c[6] := c[6] + 1; END_IF
21:       rxyz := rxyz + rxy;
22:    END_IF
23:  END_FOR
24:   out_product := rxyz;
```

**Fig. 5.5** CFG of the instrumented source code of program multiply (x, y, z)

made, along with the call parameters and highest assigned suspiciousness score. This information helps with program analysis and makes debugging faster.

To highlight and visualize the differences in the execution paths taken by the passed and failed tests, the CFG is annotated with instrumentation counter values captured as run-time measurements. The edges corresponding to execution paths taken by failed tests are highlighted with solid lines (red), passed tests with dotted lines (green), and the edges taken by both passed and failed tests with dashed lines (blue). For each program, the spectra information is then used by a suspiciousness metric to calculate and assign a suspiciousness score to the program elements.

Additionally, a thicker arrow with a line width proportional to the assigned score is employed to draw attention to the suspicious parts as per the weight. If a branching condition was executed more frequently by the failed test cases than the passed ones, the edge taken from that branching condition to the basic block is rendered with thicker edge width. This alerts the viewer that a significant number of failed test cases likely followed the path, necessitating further investigation.

For example, a control-flow graph built from the instrumented structured text code of the *multiply* program containing a statement block with the highest average suspiciousness score is shown in Fig. 5.5. As one can notice, in this case, we do not have any edges visited only by the passed tests. Based on this graph, we can conclude that the branching condition at node 16 plays an essential role in the execution of the failed tests, and it will be the starting point of the fault localization investigation.

The programs we instrumented with counter variables consist of loop-free and loop paths. To accommodate this, we normalize the number of executions of a statement inside a loop to the range of the number of tests during a test session:

$$\begin{aligned} normalised\_value = {} & (value - min\_value)/(max\_value - min\_value) \\ & * total\_number\_of\_tests\_per\_session \end{aligned} \tag{5.2}$$

In the above equation, (min_value, max_value) correspond to the range of the number of loop iterations. In essence, we use the normalized values of the counter variables in the range *(0, total number of tests per session)* to calculate the suspiciousness scores per the metrics used.

#### 5.3.2.7 Data-Flow Analysis for Suspicious Variables

In order to improve the localization of the fault, we perform the data-flow analysis of the variables involved in the most suspicious statements. All the variables used in the statements of each basic block receive the same suspiciousness score as the basic block. We analyze the suspicious variables in order to trace back the root cause of the failure. For any given variable $v$ in a program $P$, data-flow analysis [29] determines the dynamic interactions between updates of $v$ and subsequent usages of $v$ during the course of execution of $P$. A definition *(def)* of $v$ in the code denotes a statement that changes the value of *v*, and a *use* denotes the statement that uses the variable *v* in conditions, predicate use *(p-use)*, or as an argument of an assignment statement, computational use *(c-use)*. In the data-flow analysis step, suspicious variables are extracted from the statements in the detected suspicious blocks and ranked according to the frequency of def-use in those blocks. The def-use information of the suspicious variables and a heat map of the data variables mapping them to the block IDs are generated. In order to reduce the effort of the fault localization process, data analysis is only performed on the statements and variables that have a suspiciousness score. With respect to the CFG in Fig. 5.5, we will start investigating the blue and red regions, and discard the rest, as they were not visited by failed tests. Consequently, we start the investigation from the basic block on line 21, where statement `rxyz := rxyz + rxy;` is present. Both variables involved `rxyz` and `rxy` will receive a suspiciousness score of 0.74 in this block and will be solved into input parameters of the program by unfolding the loop. At the moment, this step is performed manually, but we are currently investigating ways to automate it.

#### 5.3.2.8 Fault Report

The fault report contains a synthesis of all the information generated at the previous steps, such as call and control-flow graphs, suspicious statements per program, suspicious variables, the def-use information of the suspicious variables, and a heat map of the suspiciousness score associated with potential faulty elements to help the developers in fault diagnosis and program debugging.

To summarize, the main contributions of this phase are as follows: we apply metamorphic slicing and program spectra analysis to structured text code, we use an average suspiciousness score normalized for loops for calculating suspiciousness, we generate annotated CFGs to facilitate analysis, and we perform data-flow analysis on the suspicious variables.

## 5.4 Evaluation

We have applied our approach on a *load position system* (LPS) which determines the position of a hanging load using attached markers on the hoisting frame. The LPS regularly receives up to 26 markers as [x,y] pixel coordinates from a camera module. The input may contain three markers on the hoisting frame attached to the load, as well as different light reflections in the environment (water, rain, snow, dust, etc.), which the camera filter could not remove. These reflections captured by the camera module correspond to light reflections in the water while loading containers from the vessel using a ship-to-shore (STS) crane. Only the markers corresponding to the three markers placed on the hoisting frame carrying the load are the *true markers* that determine the position of the load (see Fig. 5.6). The two markers placed on the

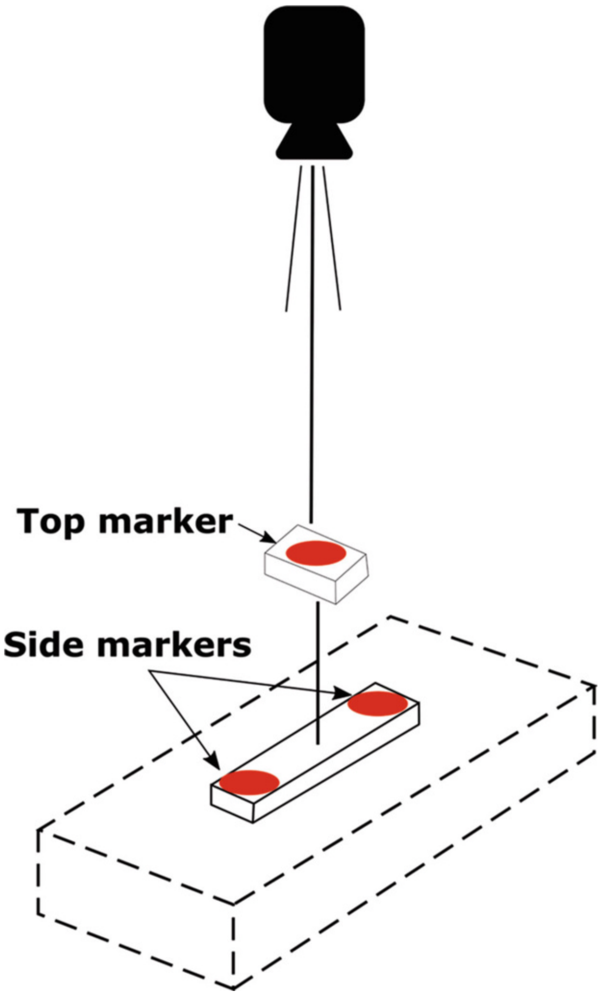


**Fig. 5.6** Positional markers in load position system

sides of the hoisting frame are referred to as *side markers*. The *top marker* is used to detect the tilt of load and to increase the probability for the algorithm to identify the true markers.

For each set of markers, the LPS tries to identify the true markers and discard the markers corresponding to reflections. The LPS produces two outputs: a Boolean value *found* indicating whether *true markers* are identified and a vector of three integers $[I_{tm_1}, I_{tm_2}, I_{tm_3}]$, indicating the index in the input marker array of the positional markers identified as *true markers*. Whenever the LPS is not able to identify the true markers consistently, the entire system can potentially move to an unsafe state and requires human intervention.

### 5.4.1 Metamorphic Testing Phase

In the above context, we map the concepts of the metamorphic testing on the LPS as follows:

#### 5.4.1.1 The Output of the LPS

$f(\{m_1, m_2, \ldots, m_n\})$ is a pair $(found, [I_{tm_1}, I_{tm_2}, I_{tm_3}])$, where $m_i$ and $tm_{i_{[1,3]}}$ are all and true positional markers respectively with two coordinates $x_i$ and $y_i$, where $\{tm_1, tm_2, tm_3\} \subseteq \{m_1, m_2, \ldots, m_n\}$, $found = TRUE|FALSE$ and $[I_{tm_1}, I_{tm_2}, and I_{tm_3}]$ are the vector of indexes of true markers provided that $found = TRUE$.

#### 5.4.1.2 Metamorphic Relation

We extract the MR from the requirements of the SUT. For instance, the following requirement "*Assuming that the system is able to classify correctly a set of markers detected by camera module in the absence of reflections (noise), the system should be able to classify correctly the same inputs in the presence of reflections*" can be formulated as the following metamorphic relation: $f(X_s) \equiv f(X_s \cup X_n)$ where $X_s$ and $X_n$ denote the series of seed input and noise markers respectively.

#### 5.4.1.3 Creating the Seed Input

In our approach, we choose the seed input as a series $X_s = \{s_1, s_2, \ldots, s_k\}$ of true marker triplets, where each element $s_i = \{tm_1^i, tm_2^i, tm_3^i\}$ has three markers which is the minimum number of markers needed for correct classification. For accuracy reasons and in order to avoid unrealistic seed input values, we extract the seed input from previous executions of the LPS by selecting those log entries that only contain three positional markers. When the seed input data set is extracted from the execution trace, we run an initial test session against the SUT to confirm that all the input marker positions are classified correctly. In case execution logs are unavailable, the seed input data can be collected from the simulation environment of the LPS that is validated against a real crane for the set of inputs the seed is extracted from.

#### 5.4.1.4 The Morphed Input

In our case, the morphing transformation takes each sample in the seed input $X_s$ and adds markers corresponding to reflections, which we denote as *noise*: a series of noise markers corresponding to environment reflections $X_n = \{n_1, n_2, n_3, \ldots, n_j\}$, where $n_i \in X$.

In the *exploration step*, we use randomly generated noise to perform an initial exploration of the SUT in order to collect observations and identify fault patterns. To this extent, we create random noise coordinate pairs of marker vectors of different lengths ranging from 1 to 23. These noise vectors are appended to the seed input one at a time. The algorithm for generating morphed input generates random (x, y) coordinates as noise with a value in the range [0, 131072], which is the size of the camera frame. Therefore, the morphed input for this step can be defined as a series $X_m = \{m_1, m_2, \ldots, m_k\}$ of sampling the markers, where each sample $m_i = \{tm_1^i, tm_2^i, tm_3^i, n_1^i, n_2^i, n_3^i, \ldots, n_j^i\}$ , $j \leq 23$, is the combination of seed input markers $X_s$ and noise markers $X_n$.

In the *exploitation step*, we analyze noise patterns in the *morphed input* that caused the system to make incorrect classifications. This led to the following observations: *the samples containing noise markers having the same geometric pattern of true markers can trigger faulty behavior of the system.* Therefore, we refine the morphed input to a more constrained version of the input space to exploit the above-mentioned fault patterns. Therefore, the refined morphed input is denoted as a series of marker samples $X'_m = \{m_1, m_2, \ldots, m_k\}$, where each sample $m_i = \{tm_1^i, tm_2^i, tm_3^i, n_1^i, n_2^i\}$ in the first follow-up test and $m_i = \{tm_1^i, tm_2^i, tm_3^i, n_1^i, n_2^i, n_3^i\}$ in the second follow-up test and $C'_m$ is the restrictive constraint used to refine the added noise to two and three noise markers. Then, the series $X'_m$ is the combination of seed input markers $X_s$ and a restricted set of noise markers $X'_n$ satisfying the constraint $C'_m$.

In order to automate the creation of the noise markers, we create replicas of the true markers, thus obtaining a similar geometrical pattern in the noise. For each sample of true markers in the seed input, we distribute the noise markers in a rectangular grid pattern in the camera frame in order to obtain a uniform distribution of samples over the input space.

In addition, each replica of the true marker placed on the grid is rotated by angles 45°, 90°, and 135° to evenly distribute the noise markers in a star-like pattern (see Fig. 5.7). Depending on the number of test inputs that we want to obtain, we can increase or decrease the number of rotations of each sample. We note that the approach is completely automated and it allows us to change the number of generated tests by changing the density of the grid and the number of rotations of the noise markers.

In our work, two sets of experiments have been conducted for each phase in the approach. For the metamorphic testing phase, we use a seed input with 625 samples, each containing a sequence of three true markers extracted from execution logs.

In the exploration step, the test generation algorithm produced $625 \times 10 \times 23 = 143{,}750$ follow-up tests. From the total of 143,750 executed tests, 143,615 satisfy the metamorphic relationship, while 135 do not. From the failed tests, 39 selected the wrong combination of inputs as true markers (false positives), whereas 96 could not find true markers among the inputs, although they were present there (false

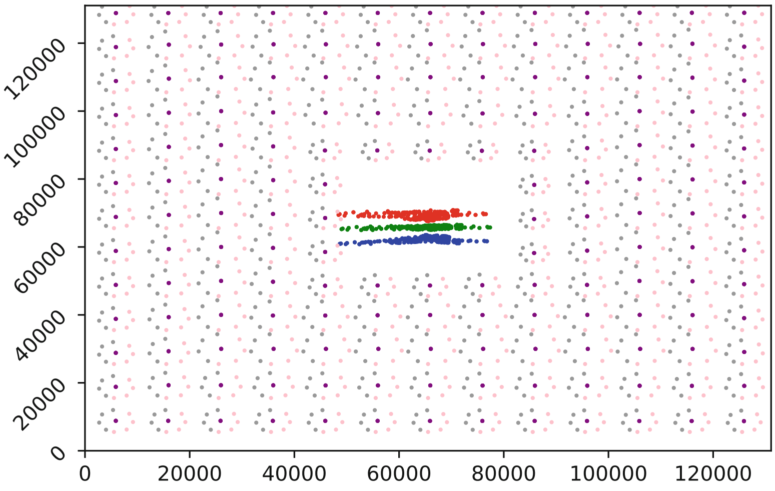


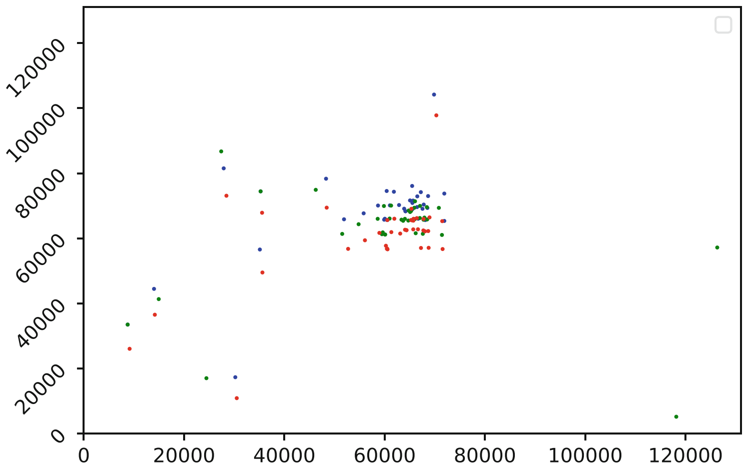
**Fig. 5.7** Test data distribution for the guided star approach

negatives). The geometric distribution of the incorrectly classified data points is shown in Fig. 5.8.

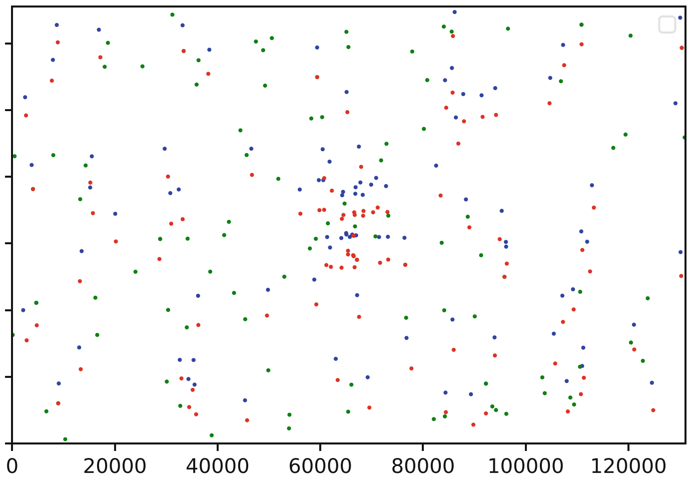
The geometric distribution of incorrectly classified data points is shown in Fig. 5.8. Further analysis of the failed tests provides us with the following observations. FP results occurred when the input set contained either a set of *two noise markers* resembling the pattern of true side markers or a set of *three noise markers* resembling the geometrical pattern of the true marker triplet. FN results occurred when the input set contained either a number of markers greater than or equal to 6 or a set of *two noise markers* resembling the pattern of the true side markers.

In the exploitation step, we ran two separate testing sessions in which the refined morphed input contained two and three noise markers, respectively, besides the true markers. In both cases, the test generation algorithm produced 2400 refined morphed test inputs. These refined morphed inputs were created by replicating and rotating the seed input markers. It resulted in $625 \times 4 = 2500$ noise markers and discarding the samples that do not fit in the $131072 \times 131072$ frame after the *rotate* morphing action. The results of the test execution are shown in Table 5.4. For the test session with five input markers, 7 incorrect and 74 missed classifications have been identified, whereas, for the subsequent test with six input markers, no incorrect/missed classification occurred.

For the test session with 5 input markers, 7 FP and 74 FN classifications are identified, whereas, for the subsequent test with 6 input markers, no FN and 92 FP classifications are identified. The test execution results of the guided method, where the number of markers is five, contain more FNs indicating that the system is not identifying the true markers when a replica of two side markers is added as noise. The distribution of the noise markers in the input space for failed tests (FNs) in the exploitation phase is shown in Fig. 5.9.



(a)



(b)

**Fig. 5.8** Morphed input corresponding to incorrect classifications in the exploration phase. (**a**) Morphed input w.r.t FP output. (**b**) Morphed input w.r.t FN output

However, the FP results of the follow-up test where the number of markers is six reveal that a replica of three true markers can trigger an incorrect identification and compromise the functional safety of the system. The distribution of the noise markers in the input space for failed tests (FPs) in the exploitation phase is shown in Fig. 5.10. It is also observed that a replica of two side markers has a low chance of causing FPs compared to the noise created with a replica of three true markers. Moreover, only the noise markers corresponding to the exact replica of true markers

**Table 5.4** Test execution results

| Method | No. of markers | No. of tests | TPs | FPs | FNs | FDR |
|---|---|---|---|---|---|---|
| Exploration | 4–26 | 143750 | 143615 | 39 | 96 | 0.0009 |
| Exploitation | 5 | 2400 | 2319 | 7 | 74 | 0.03 |
| | 6 | 2400 | 2308 | 92 | – | 0.04 |

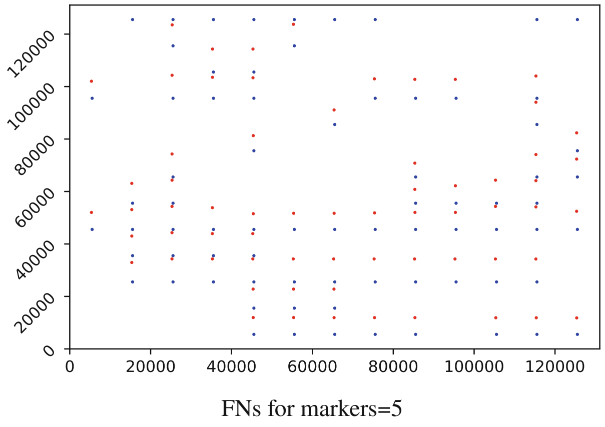


**Fig. 5.9** Input distribution for failed tests (FN) in the exploitation phase

triggered the incorrect identification of true markers. In addition, we can observe that the noise markers rotated by angles $45^0$, $90^0$, $135^0$ resulted in TP test cases, where the system correctly identified the true markers despite the noise.

Table 5.4 also shows the corresponding fault detection ratio (FDR) [14] for each phase as the number of tests that found a fault in the entire tests suite. As expected, the exploration phase has a very low FDR due to the random test generation, whereas in the exploitation phase, FDR has increased around 33 to 44 fold.

The correct classification of true markers, in the morphed output, satisfies the MR and counts as tests that do not fail. The failed tests include incorrect identification or missed identification of true markers placed in the first three positions in the morphed input.

## 5.4.2 *Fault Localization Phase*

As previously discussed, the goal of the fault localization phase is to identify the code statements or variables which are the root cause of the failed tests observed in the metamorphic testing phase. To this extent, we have to compare the execution traces of the passed tests against the execution traces of the failed tests. In the previous phase, we observed two types of incorrect classifications: false positives

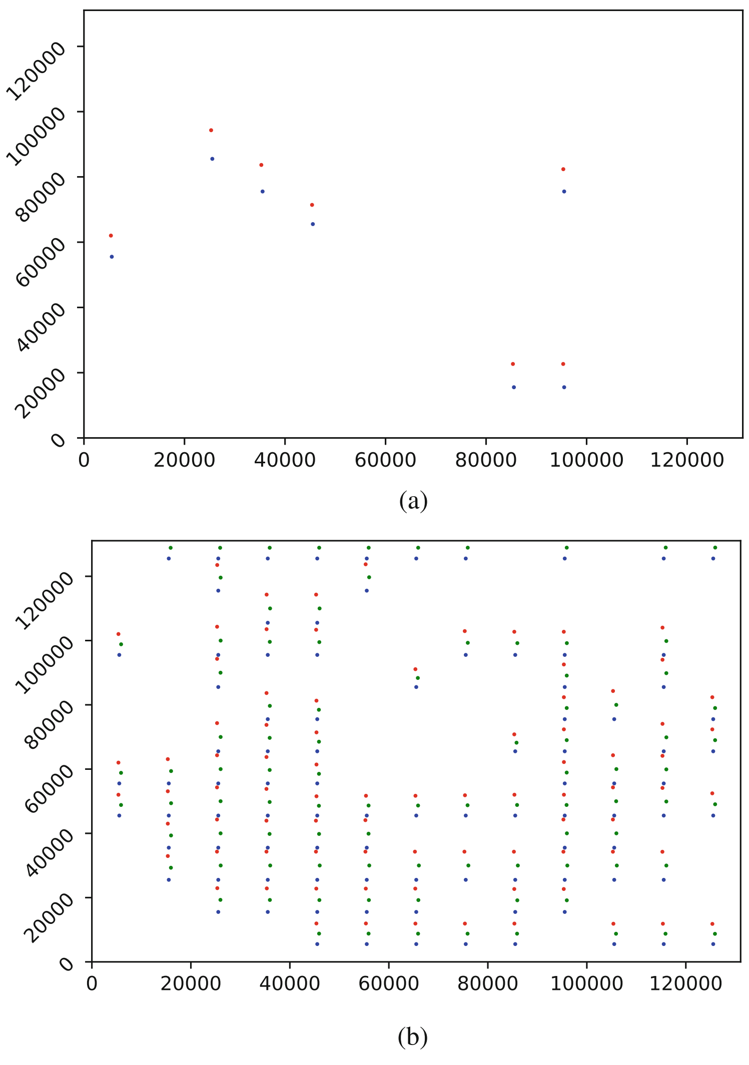


**Fig. 5.10** Input distribution for failed tests (FP) in the exploitation phase. (**a**) FPs for markers = 5. (**b**) FPs for markers = 6

and false negatives. In order to make the localization more precise, we compare the execution traces of the tests of one type of failure against the execution traces of passed tests. The same process can be applied to the other types of failures in different localization sessions.

In our case, we have randomly selected a number of tests from the false positive ones and the same number from the passed ones. For instance, a test suite is generated using the noise marker coordinates from the passed tests in the exploration phase and another using the fault-inducing noise markers identified from failed tests in the exploitation phase. We keep the number of markers in each test and the number of tests in each test suite to be the same as discussed in Sect. 5.3.2.1. It ensures a fair comparison between the test suites w.r.t to the suspiciousness score calculated for the execution paths.

For instrumentation purposes, we extract the structured text code of the LPS. The LPS is written in the PLC programming language function block diagram (FBD) and structured text (ST). It comprises of inter-procedural calls between function blocks and functions written in ST, similar to the structure depicted in Fig. 5.5.

We instrumented the code with counters in each program and executed the tests. In total, we have added six counter arrays, each of length corresponding to the number of branches in the program. We have selected different sizes of the passed and failed test suites, namely, 10, 20, and 50 tests, to run the fault localization. Each test session was run five times for statistical purposes. The scores calculated using the three formulas slightly varied since different test suites had slightly different traces in the program, but they still pointed to the same fault, as discussed in the following. Additionally, the spectra information collected from the test execution sessions and the calculated suspiciousness score are annotated on the control-flow graph generated per program in the fault report, similar to the generic examples in Figs. 5.11 and 5.5. The suspiciousness score annotated in the graph at the program level is the maximum of the scores assigned at the basic block level for each program.

For experimentation purposes, we have used the following three suspiciousness metrics Ochiai, Jaccard, and Tarantula, as well as the average suspiciousness score defined in the previous section. Table 5.5 summarizes the suspicious variables that appear most often in the basic blocks and branching conditions identified by the suspiciousness metrics used when using test suites of size 10. For confidentiality purposes, the names of variables have been anonymized, and as such, Table 5.5

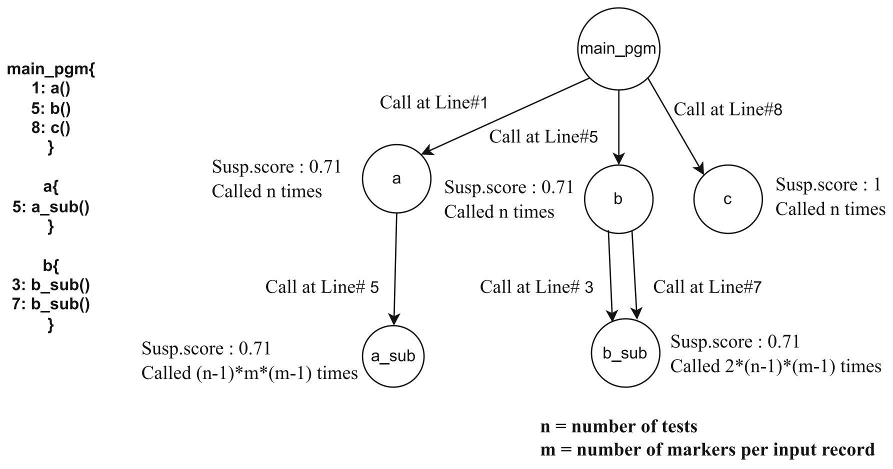


**Fig. 5.11** The call graph of the LPS

**Table 5.5** Suspiciousness score for variables/blocks in programs under test for test suites of size 10

| Suspicious variable | Block ID | Program name | Scores | | | |
|---|---|---|---|---|---|---|
| | | | Ochiai | Jaccard | Tarantula | Average score |
| var_in_ss1 | 8 | a | 0.71 | 0.5 | 0.5 | 0.57 |
| | 10 | a | 0.71 | 0.5 | 0.5 | 0.57 |
| var_out_a_ss1 | 10 | a | 0.71 | 0.5 | 0.5 | 0.57 |
| | 14 | a | 0.71 | 0.5 | 0.5 | 0.57 |
| | 17 | a | 0.71 | 0.5 | 0.5 | 0.57 |
| var_out_a_ss2 | 10 | a | 0.71 | 0.5 | 0.5 | 0.57 |
| | 14 | a | 0.71 | 0.5 | 0.5 | 0.57 |
| | 19 | a | 0.71 | 0.5 | 0.5 | 0.57 |
| var_out_a_sub_ss | 1 | a_sub | 0.71 | 0.5 | 0.5 | 0.57 |
| | 3 | a_sub | 0.71 | 0.5 | 0.5 | 0.57 |
| | 7 | a_sub | 0.71 | 0.5 | 0.5 | 0.57 |
| | 2 | b | 0.71 | 0.5 | 0.5 | 0.57 |
| | 4 | b | 0.71 | 0.5 | 0.5 | 0.57 |
| var_out_b_ss1 | 2 | b | 0.71 | 0.5 | 0.5 | 0.57 |
| | 3 | b | 0.70 | 0.54 | 0.70 | 0.65 |
| | 6 | b | 0.71 | 0.5 | 0.5 | 0.57 |
| | 9 | b | 0.71 | 0.5 | 0.5 | 0.57 |
| var_out_b_ss2 | 4 | b | 0.71 | 0.5 | 0.5 | 0.57 |
| | 5 | b | 0.34 | 0.2 | 0.29 | 0.27 |
| | 6 | b | 0.71 | 0.5 | 0.5 | 0.57 |
| | 11 | b | 0.71 | 0.5 | 0.5 | 0.57 |
| var_out_b_sub_ss | 1 | b_sub | 0.71 | 0.5 | 0.5 | 0.57 |
| | 3 | b_sub | 0.71 | 0.5 | 0.5 | 0.57 |
| | 7 | b_sub | 0.71 | 0.5 | 0.5 | 0.57 |
| | 9 | b_sub | 0.71 | 0.5 | 0.5 | 0.57 |
| | 11 | b_sub | 0.71 | 0.5 | 0.5 | 0.57 |
| | 15 | b_sub | 0.71 | 0.5 | 0.5 | 0.57 |
| | 19 | b_sub | 0.57 | 0.36 | 0.8 | 0.57 |
| | 21 | b_sub | 0.32 | 0.1 | 1 | 0.47 |
| var_in_a_ss1 | 7 | c | 0.67 | 0.47 | 0.5 | 0.54 |
| **var_in_a_ss2** | **11** | **c** | **0.89** | **0.8** | **1** | **0.89** |
| var_in_b_ss1 | 15 | c | 0.67 | 0.47 | 0.5 | 0.54 |
| **var_in_b_ss2** | **19** | **c** | **0.89** | **0.8** | **1** | **0.89** |
| var_out_c_ss | 52 | c | 0.67 | 0.47 | 0.5 | 0.54 |

shows the masked names of the variables which correspond to the suspicious variables in the programs under test. In the next step, these variables whose values are defined and used repeatedly in suspicious blocks and their def-use chains in the control flow and inter-procedural flow are subjected to data-flow analysis. As one can notice, the average score indicates the same ranking as the other three suspiciousness metrics.

To that extent, all suspicious variables extracted from the basic blocks are subjected to a data-flow analysis to improve the precision of the fault localization. A heat map for the suspicious variables extracted from suspicious blocks in Table 5.5 based on their *def-use* chain is created as shown in Fig. 5.12. Data-flow analysis of these variables based on their *dynamic_mslice* and *def-use* chain reveals the variable which propagates the error triggering the fault.

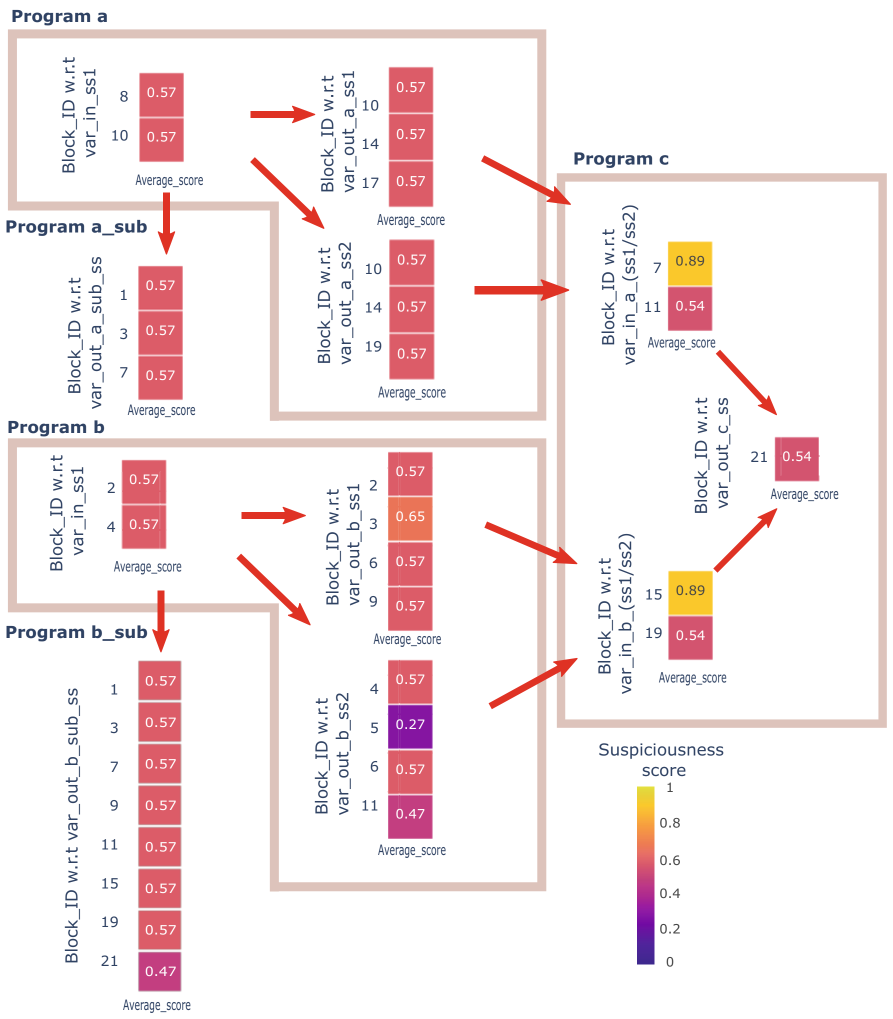


**Fig. 5.12** Heat maps revealing def-use chain of suspicious variables on the call graph of the LPS in Fig. 5.11

In this case, variables *var_in_a_ss2* and *var_in_b_ss2* in program *c* appeared in the basic blocks with maximum suspiciousness score indicated by all the metrics used in this experiment. The data-flow related to these variables was analyzed further to determine their def-use chain in the inter-procedural call structure. The *dynamic mslice* extracted for these variables revealed that the value of these variables was defined in program *a* and program *b*, respectively, based on the calls to subprograms *a_sub* and *b_sub*. According to the data-flow analysis stated above and tracking the definition and usage of variables associated with the identified suspicious blocks, we reach the variable *var_in_ss1*, an input variable in program *a* and program *b*. This variable *var_in_ss1* was used in the branching condition that impacts the definition of the variables *var_out_a_ss2* and *var_out_b_ss2* whose value was propagated in the program flow and appear in program *c* as the most suspicious variables *var_in_a_ss2* and *var_in_b_ss2*. Based on these findings, *var_in_ss1* was suspected of significantly impacting the program flow in terms of influencing the def-use chain of the suspicious variables listed in Table 5.5. This study revealed how the definition and usage of suspicious variables were propagated in the program flow to induce the fault and result in the incorrect final state.

Upon inspection of the data flow of the *var_in_ss1* variable, we discovered that it was using a hard-coded value instead of using the value assigned as an input while calling program *a* and program *b*. Such a fault has direct implications on the safety of the system, which, if properly implemented, will move to a safe state when the input classification is not successful. However, this fault may also be considered a security vulnerability according to Common Weakness Enumeration database, CWE-547: use of hard-coded, security-relevant constant.[1]

In addition, in the presence of this vulnerability, the LPS is not able to filter out incorrect (noisy) input which is consistent with another type of security vulnerability, namely, CWE-20: improper input validation.[2] Since the LPS is connected to other components via a network, an attacker can inject "noise" into the input of the LPS, which can affect the classification of the markers, either sending the system to an unsafe state or providing false information about the position of the load.

Upon updating the assignment of this variable and re-executing the failed test suite, all the previous metamorphic tests passed. And the test results also proved that the system no longer exhibits faulty behavior in the follow-up tests.

Table 5.6 shows how the proposed method of metamorphic program slicing and data slicing based on def-use chain analysis reduces the scope of data-flow analysis to 35% of the entire input space. Based on these results, the integrated method proposed in this work considerably reduced the complexity and effort in fault diagnosis and program debugging. Therefore, it shows the applicability and effectiveness of the technique in the chosen case study.

To evaluate the benefits and effectiveness of the proposed approach, we have also conducted a controlled experiment involving 20 software professionals as

[1] https://cwe.mitre.org/data/definitions/547.html

[2] https://cwe.mitre.org/data/definitions/20.html

**Table 5.6** Reduction in scope of search using the metamorphic fault localization approach

| Phase | Reduction in scope of search |
|---|---|
| Code analysis | 233/701 LoC |
| Code analysis | 65/133 basic blocks |
| Data-flow analysis | 60/170 variables |

subjects. They were divided into two balanced groups with an equal distribution of programming expertise and experience level. The first group received the source code of a program for triangle classification having 27 lines of code. The code had an inserted fault in it and was accompanied by a set of passed and failed metamorphic tests. The second group received the same source code, the passed and failed tests, and the annotated CFGs that highlighted the suspicious parts in the code. The second group also received preliminary training on how the annotated CFGs should be interpreted. Each group had 15 minutes to find the inserted fault using the available material. The second group was able to identify faster the fault in the code in an average of 8 minutes, while for the first group, only one person localized the fault before the end of the experiment. After the experiment, the first group also received the same training on how to use the annotated control-flow graph. The participants in both groups overwhelmingly agreed that having the annotated CFGs greatly facilitates the fault localization process.

### 5.4.3 Tool Support

Most steps of the approach are fully automated. For instance, in the metamorphic testing phase, the test generation for the exploitation and exploration phases is fully automated. Only the formalization of the metamorphic relation and the extraction of the fault-revealing patterns require human intervention. In the fault localization phase, program extraction, instrumentation, test execution, score calculation, and graph generation are also fully automated. Currently, only the data-flow analysis requires manual effort. We plan to automate the inter-procedural data-flow analysis by extracting and resolving the dependencies of suspicious variables in different subprograms.

To implement automation, we have used Python and the Python testing framework `Pytest`, which interacts with the SUT to perform automated test execution of the metamorphic and fault localization phases. The application program runs on CODESYS SoftPLC V3.5 with an in-built OPC UA server that enables the communication between the CODESYS OPC UA server [30] and the Python OPC UA client [31] via the OPC UA [32] data exchange protocol as shown in Fig. 5.13. The run-time measurements, such as code coverage and executable lines of code for the programs under test, were measured using the CODESYS Profiler.

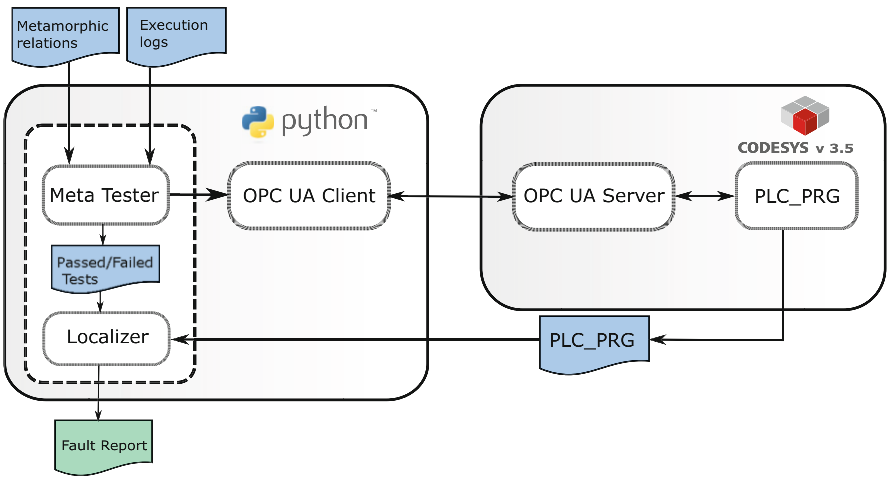


**Fig. 5.13** Tool chain for metamorphic test generation and fault localization

## 5.5 Threats to Validity

In this section, we discuss the possible threats to the validity of the results of our study.

### 5.5.1 *Construct Validity*

The current work has not been validated against time-critical systems. Our experiments are applied to a PLC application written in FBDs and structured text with no timer functions. It is possible that the approach may collect less accurate execution information in the presence of timer-dependent variables. However, the system in our study is a real-world ICS in PLC programming language for which the proposed method using OPC UA as a communication protocol is standardized under Industry 4.0 for its technical interoperability [33].

### 5.5.2 *External Validity*

The fault detection and localization effectiveness reported in the study are based on the chosen metamorphic relation for the system under test. The results reported from the experiments are based on the input variables defined in the input MR and output behavior, and the test verdict relies on those. Like any other testing method, the MT is not exhaustive, even if the MR has been identified. For ICSs with large input space, different MRs can be identified. The choice of MRs that uses a different set of input variables may influence the outcome of both the metamorphic testing and fault localization phase. Therefore, the approach's effectiveness for any industrial software system with a test oracle problem may vary depending on the metamorphic relations that can be defined for it.

### *5.5.3 Conclusion Validity*

Based on current experimental evaluations, we noticed that the average score of the suspiciousness metrics Ochiai, Jaccard, and Tarantula we proposed gives a good perspective on the spectrum-based fault localization. However, more investigations are needed to evaluate the risk of imprecision while using the proposed score in pinpointing the fault location for other industrial software systems.

## 5.6 Related Work

The metamorphic testing technique has been successfully applied in several application domains for testing software systems with an oracle problem. Among the most popular ones are Web services and applications [34–36], embedded systems [37–39], simulation and modeling [40–42], computer graphics [43, 44], and various other domains.

In a study by Wang et al. [45], a metamorphic testing system called `METAOD` designed for deep learning-based object detectors (ODs) is used for identifying objects in an image using neural networks. The `METAOD` system is used for object extraction, selection, and insertion, to test the image classification accuracy of the object detectors. The object extraction module of `METAOD` takes a set of images as input and extracts object instances using segmentation techniques. Such synthetic images with inserted objects are used to test the ODs to evaluate their prediction accuracy. The synthetic images that triggered erroneous predictions were used to retrain the model and improve its accuracy. Even though the techniques are different, the test data generation by noise insertion, also used in our study, is similar to the synthetic image generation method in `METAOD`. However, our work focuses on systematically generating morphed input from failure-inducing patterns discovered in source and follow-up test executions.

Xie, Xiaoyuan, et al. [25] introduced the concept of metamorphic slicing by integrating metamorphic testing and program slicing in combination with SBFL. They conducted an experimental study on nine programs of varying sizes using three MRs for each. They used three risk evaluation formulas, Jaccard, Ochiai, and Tarantula, to show the effectiveness of the approach in fault localization and the practical applicability of the technique in applying SBFL for application programs with an oracle problem. Their study also identified two faults in the chosen programs and proved that the approach of using violation or non-violation of a metamorphic test group as a test oracle for SBFL achieved a performance level quite similar to conventional SBFL techniques while testing systems with test oracle. Similarly, we combine metamorphic testing with SBFL and program slicing techniques.

In his study on program debugging, Zeller, A [46] discussed program slicing based on the control-flow and data-flow dependencies to form the program-dependence graph. Their approach of slicing programs based on dependencies could detect code smells such as unreachable code, uninitialized variables, and unused values, thereby assisting in failure diagnosis by reducing the scope of analysis to deduce the cause of program failure. However, our work has the additional step of integrating data-flow analysis with a definition-use chain associated with inter-procedural calls for fault localization.

A recent survey about challenges and opportunities in metamorphic testing [26] discusses the possibility of extending the integration of MT and SBFL to wider application domains that face the oracle problem. Owing to the complexity of such systems that might include inter-procedural program structure, they are also categorized as hard to debug and verify for any underlying code vulnerabilities. In our work, we focus on the applicability and effectiveness of this integrated method capitalizing on usability features for industry scale control software testing facing an oracle problem and hence being commonly considered non-testable. We also provide several improvements, such as using normalized values of the counters for evaluating loops, analyzing data-flow information, and using several metrics for suspiciousness score calculation.

## 5.7 Conclusions and Future work

We proposed a metamorphic testing approach for PLC-based ICSs integrated with a fault localization technique. The main contributions of our approach were the following:

1. A two-phase metamorphic testing approach comprises an exploration phase in which we learn about fault patterns of the system under test and an exploitation phase where the observed fault patterns are used for targeted testing.
2. Fault localization is based on the results of the metamorphic tests. It combines the spectrum-based fault localization and program slicing technique integrated with inter-procedural control-flow analysis and data-flow analysis for PLC programs.
3. Tool support for metamorphic test generation, execution, and fault localization in PLC programs.

The presented approach is generic and applicable to systems for which the source code is available and written in an imperative programming language and for which metamorphic relations can be identified. For validation purposes, we conducted our experiment on a PLC program unit comprising four function blocks and two functions written in structured text. The fault analysis focuses only on the fault type discovered using the MR defined in the metamorphic testing phase. Hence the empirical analysis and conclusions made are proven effective on this experimental setup, programs, and the size of the test suite chosen for this study. It is also to be noted that the fault localization performance reported in this study is based on only one MR defined in the metamorphic testing phase. A different set of MRs with varying test inputs may result in different results in the fault localization phase. Future work will also investigate applying heuristic methods for the selection of fault-inducing inputs for defining MRs.

The main advantages of this approach, in addition to alleviating the oracle problem, are providing a tool-based approach for fault localization and program debugging to assist the developers with root cause analysis and future regression testing of ICS. The study presented in this paper proves that metamorphic testing combined with SBFL can considerably reduce the effort in program debugging and program repair of real-time ICS. The identification of a metamorphic relation is done manually based on the specification of the system. A known challenge in the identification of MRs is the need for domain expertise to assess the expected input and output behavior of the system. Therefore, it is of interest to conduct a comparative study on the findings of fault localization and its effectiveness in terms of fault localization performance based on the choice of MRs and different types of metamorphic groups of inputs. As a future work, we plan to automate the identification of MR for an ICS from its specification and explore the applicability of metamorphic and mutation-based approaches for testing ICS and apply heuristic techniques for minimization of test suites. We also plan to improve the automation of the fault localization phase using machine-learning methods and thus reduce the need for manual analysis.

**Acknowledgments** This work has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No. 957212 (VeriDevOps).

## References

1. L. Piètre-Cambacédès, C. Chaudet, Int. J. Crit. Infrastruct. Prot. **3**(2), 55 (2010). https://doi.org/10.1016/j.ijcip.2010.06.003. https://www.sciencedirect.com/science/article/pii/S1874548210000247
2. G. Kavallieratos, S. Katsikas, V. Gkioulos, Future Internet **12**(4), 65 (2020)
3. B. Lim, D. Chen, Y. An, Z. Kalbarczyk, R. Iyer, in *2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC)* (IEEE, 2017), pp. 205–210
4. R. Sun, A. Mera, L. Lu, D. Choffnes, in *2021 IEEE European Symposium on Security and Privacy* (IEEE Computer Society, Los Alamitos, CA, USA, 2021), pp. 385–402. https://doi.org/10.1109/EuroSP51992.2021.00034. https://doi.ieeecomputersociety.org/10.1109/EuroSP51992.2021.00034
5. A. Serhane, M. Raad, R. Raad, W. Susilo, in *2018 International Conference on Computer and Applications (ICCA)* (2018), pp. 348–352. https://doi.org/10.1109/COMAPP.2018.8460287
6. IEC, IEC 62443 Security for Industrial Automation and Control Systems (IACS) part 4–2: Technical Requirements for IACS components. Standard, International Electrotechnical Commission (IEC), Geneva, CH (2019)

7. V&V IEC-61508-3:2010 (2021). https://assets.vector.com/cms/content/products/VectorCAST/Docs/Whitepapers/English/Understanding_Verification_Validation_of_Software_Under_IEC-61508_v2.0.pdf
8. S.E. Valentine, PLC code vulnerabilities through SCADA systems. Ph.D. thesis, University of South Carolina, USA (2013). AAI3561883
9. A. Zeller, Computer **34**(11), 26 (2001)
10. W.E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa, IEEE Trans. Softw. Eng. **42**(8), 707 (2016). https://doi.org/10.1109/TSE.2016.2521368
11. G. Sudheerbabu, T. Ahmad, F. Sebek, D. Truscan, J. Vain, I. Porres, in *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)* (2022), pp. 1–4. https://doi.org/10.1109/ETFA52439.2022.9921439
12. T.Y. Chen et al., arXiv preprint arXiv:2002.12543 (2020)
13. H. Liu et al., in *12th International Conference on Quality Software* (IEEE, 2012), pp. 59–68
14. S. Segura et al., IEEE Trans. Softw. Eng. **42**(9), 805 (2016)
15. R. Abreu, P. Zoeteweij, R. Golsteijn, A.J. Van Gemund, J. Syst. Softw. **82**(11), 1780 (2009)
16. T. Reps, T. Ball, M. Das, J. Larus, in *Software Engineering—Esec/Fse'97* (Springer, 1997), pp. 432–449
17. M.J. Harrold, G. Rothermel, K. Sayre, R. Wu, L. Yi, Softw. Test. Verif. Reliab. **10**(3), 171 (2000)
18. Q.I. Sarhan, Á. Beszédes, IEEE Access **10**, 10618 (2022)
19. T. Wu, Y. Dong, M.F. Lau, S. Ng, T.Y. Chen, M. Jiang, Appl. Sci. **10**(1), 398 (2020)
20. M. Weiser, Commun. ACM **25**(7), 446 (1982)
21. M. Weiser, IEEE Trans. Softw. Eng. **8**(4), 352 (1984)
22. H. Agrawal, J.R. Horgan, S. London, W.E. Wong, in *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95* (IEEE, 1995), pp. 143–151
23. M. Harman, R. Hierons, Softw. Focus **2**(3), 85 (2001)
24. B. Xu, J. Qian, X. Zhang, Z. Wu, L. Chen, ACM SIGSOFT Softw. Eng. Notes **30**(2), 1 (2005)
25. X. Xie, W.E. Wong, T.Y. Chen, B. Xu, Inf. Softw. Technol. **55**(5), 866 (2013)
26. T.Y. Chen, F.C. Kuo, H. Liu, P.L. Poon, D. Towey, T. Tse, Z.Q. Zhou, ACM Comput, Surv. **51**(1), 1 (2018)
27. S. Segura et al., IEEE Trans. Softw. Eng. **44**(11), 1083 (2017)
28. Y. Wang, Z. Huang, B. Fang, Y. Li, IEEE Access **6**, 8925 (2018)
29. M.S. Hecht, *Flow Analysis of Computer Programs* (Elsevier Science Inc., 1977)
30. CODESYS OPC UA (2020). https://www.codesys.com/products/codesys-runtime/opc-ua.html
31. O. Roulet-Dubonnet, Python OPC-UA Documentation (2021). https://python-opcua.readthedocs.io/en/latest/index.html
32. S.H. Leitner, W. Mahnke, ABB Corporate Research Center **48**(61–66), 22 (2006)
33. M. Schleipen, S.S. Gilani, T. Bischoff, J. Pfrommer, Procedia Cirp **57**, 315 (2016)
34. W.K. Chan et al., Int. J. Web Serv. Res. **4**(2), 61 (2007)
35. C.A. Sun et al., in *2011 IEEE International Conference on Web Services* (IEEE, 2011), pp. 283–290
36. Z.Q. Zhou et al., IEEE Trans. Softw. Eng. **42**(3), 264 (2015)
37. F.C. Kuo et al., in *2011 IEEE 36th Conference on Local Computer Networks* (IEEE, 2011), pp. 291–294
38. T.Y. Chen et al., J. Syst. Softw. **116**, 177 (2016)
39. U. Kanewala et al., Softw. Test. Verif. Reliab. **26**(3), 245 (2016)
40. J. Ding et al., in *Proceedings of the 6th International Workshop on Automation of Software Test* (2011), pp. 1–7
41. A. Núñez et al., J. Grid Comput. **10**(1), 185 (2012)
42. P.C. Cañizares et al., Procedia Comput. Sci. **51**, 2804 (2015)
43. W.K. Chan et al., Softw. Test. Verif. Reliab. **20**(2), 89 (2010)

44. T. Jameel et al., in *2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)* (IEEE, 2015), pp. 1–6
45. S. Wang, Z. Su, in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (IEEE, 2020), pp. 1053–1065
46. A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging* (Elsevier, 2009)

# Chapter 6
# Interactive Application Security Testing with Hybrid Fuzzing and Statistical Estimators

**Ramon Barakat, Jasper von Blanckenburg, Roman Kraus, Fabian Jezuita, Steffen Lüdtke, and Martin A. Schneider**

**Abstract** Both static analysis and dynamic analysis are methods to identify vulnerabilities in programs. Whereas sound static analysis is strong in identifying all vulnerabilities of a certain type by analyzing all program paths, it suffers from high numbers of false positives which can make this approach infeasible for large amounts of code. In contrast, dynamic analysis, in particular fuzzing, has a low number of false positives but suffers from the inability to prove the absence of bugs since it covers only specific execution paths. Therefore, many bug-triggering paths may not be executed. This can then lead to potentially high numbers of false negatives, i.e., missing observations of bugs which are actually present in the code. Since both methods have complementary strengths and weaknesses, interactive application security testing (IAST) aims at obtaining the best from both methods by a smart and interactive combination to mutually eliminate the weaknesses of each method. For instance, fuzzing techniques can be used to discriminate the true positives and the false positives from the static analysis, and static analysis can benefit from concrete values observed during test execution to make the analysis more precise. However, interactive application security testing comes with its own challenges that need to be solved using a set of methods and techniques. In this chapter, we present an approach to both automatically assess static analysis results using fuzzing to make static analysis feasible for large-scale projects and to improve fuzzing with results from static analysis, e.g., by using results from constant propagation, such as magic bytes, to cover code fragments that are hard to reach for traditional fuzzers.

**Keywords** Interactive application security testing · Hybrid fuzzing · Constraint solving · Static analysis · Statistical methods

R. Barakat · J. von Blanckenburg · R. Kraus · F. Jezuita · S. Lüdtke, · M. A. Schneider (✉)
Fraunhofer Institute for Open Communication Systems (FOKUS), Berlin, Germany
e-mail: Ramon.Barakat@fokus.fraunhofer.de; Jaspervon.Blanckenburg@fokus.fraunhofer.de; Roman.Kraus@fokus.fraunhofer.de; Fabian.Jezuita@fokus.fraunhofer.de; Steffen.Ludtke@fokus.fraunhofer.de; Martin.Schneider@fokus.fraunhofer.de

A. Sadovykh et al. (eds.), *CyberSecurity in a DevOps Environment*,
https://doi.org/10.1007/978-3-031-42212-6_6

## 6.1 Introduction

Cybersecurity attacks on software systems are part of our daily lives. Nearly every day, new incidents with dramatic consequences are reported, leading to security breaches up to the unavailability of the IT infrastructure of entire organizations. Many of such attacks are possible due to the popularity of programming languages that do not provide inherent memory safety features, such as C [1]. Even though such vulnerabilities are known for more than two decades, they still persist in today's software products and pose a significant security problem, comprising nearly 15 % of all vulnerabilities in 2021 [2]. One of the most famous vulnerabilities was the [1] that is located in the widely used OpenSSL library and enables an adversary to read confidential data from the memory, including cryptographic keys and passwords. Despite its severity[2] and simplicity, it remained undiscovered for more than 2 years.

Due to its popularity, much research has been spent on techniques to identify such vulnerabilities. An established Dynamic Application Security Testing (DAST) technique to identify them is fuzzing. In fuzzing, the System Under Test (SUT) is stimulated with random inputs. Despite its simplicity, this technique is very effective. On the contrary side, static analysis techniques, belonging to Static Application Security Testing (SAST), analyze the source code of a program for patterns that hint at a vulnerability. Both techniques SAST and DAST have their strengths and drawbacks. However, applying these techniques separately did not allow to completely overcome memory-related vulnerabilities as proven in the past.

Even though research on combining static and dynamic analysis, known as Interactive Application Security Testing (IAST), started several years ago, there is still no methodology in which both approaches interact in both directions, i.e., in which DAST is benefiting from SAST and, at the same time, SAST benefiting from DAST, to reliably identify memory-related vulnerabilities with nearly no manual interventions.

In this chapter, a novel approach of hybrid fuzzing is presented, where static and dynamic analyses are used to uncover vulnerabilities in a targeted manner and augmented with statistical means, aiming at identifying not only true but also false positives. We develop as part of a research project a novel approach from the perspective of the dynamic analysis by using the results from the static analysis. The remainder of this chapter describes our methodology for hybrid fuzzing as an example of an IAST approach and discusses the following research questions that have been identified along the research that led to the proposed methodology:

[1] https://heartbleed.com/

[2] It has a CVSS score of 7.5 out of 10, cf. Common Vulnerability Scoring System, https://nvd.nist.gov/vuln-metrics/cvss

**RQ1** *Which information can static analysis provide to DAST that facilitates its analysis?*
This research question is related to the kind of interaction of SAST and DAST. Our hypothesis is that static analysis can guide the dynamic analysis where in the code to search for vulnerabilities, and can provide information that enables targeted testing for these vulnerabilities.

**RQ2** *Is IAST more efficient than DAST on its own? Under which conditions is IAST more efficient than DAST?*
In other words, is IAST worth the additional effort to stand alone DAST, i.e., is it worth putting more effort in generating crafted test cases, or is brute-force testing as done by black box and gray box fuzzers more efficient? Our hypothesis here is that IAST plays its advantage when the SUT is well tested and contains only a few, deeply hidden bugs, where complex path constraints constitute a natural barrier for traditional DAST.

**RQ3** *To which extent can false and true positives from static analysis be automatically discriminated by dynamic analysis?*
This is related to the challenge of identifying false positives from the static analysis through dynamic analysis that cannot prove the absence of bugs [32]. Hence, we hypothesize that we can verify true positives quite well, but doing this for false positives is much harder, since we cannot solely rely on the tests itself and need in addition to rely on statistical methods.

**RQ4** *How well do methods used in DAST to quantify the uncertainty to discover new bugs work in the context of IAST, in particular when we aim at verifying SAST results using DAST?*
This research question deals in particular with the false positives and, thus, is closely related to RQ3. Statistical estimators such as the Good-Turing estimator have already been proposed in traditional gray box fuzzing to quantify uncertainty. We hypothesize that it is also suitable in the context of IAST.

## 6.2 Related Work

Dynamic application security testing (DAST) aims to analyze and identify bugs in an SUT by executing it. DAST includes various techniques, the most prominent of which is fuzzing [3, 4]. In fuzzing, the SUT is stimulated with invalid or unexpected inputs to, for instance, crash it and, thus, identify potential security issues. The underlying assumption of fuzzing is that if input validation mechanisms in the interface of an SUT are faulty or even missing, invalid and unexpected inputs might bypass them and, thus, may alter the application logic and bypass security controls. This may result in memory corruption that may allow an attacker to crash a system or to inject malicious code. Complementary to this approach is behavioral fuzzing, which can detect errors in the processing of calls, implemented state machines, and call sequences [5, 6].

In general, fuzzing approaches can be distinguished by their aim to maximize code coverage, to target a specific subset of the code, or to identify specific vulnerabilities. Some of these approaches employ static analysis to cover code fragments that are hard to reach, e.g., because they are deeply nested or hidden behind complex path conditions.

While early approaches simply generate data randomly, model-based fuzzers were considered more powerful due to their ability to generate semi-valid input data, i.e., input data that deviates slightly from valid input data and, thus, challenges the input validation. With the success of American Fuzzy Lop (AFL) [7] and LibFuzzer [8], fuzzing with random mutation of input data has returned, which could be easily deployed since it does not require any protocol knowledge. However, even though input data is generated randomly, the power of these tools arises from employing search-based methods, especially genetic and evolutionary algorithms, which use mutation, recombination, and selection to mimic natural evolution and, thus, to find those inputs that would execute new parts of the SUT's code while aiming at maximizing the code coverage.

Even though many different vulnerabilities have been uncovered using AFL,[3] its random-based mutation is not very efficient. Nested branches, multi-byte markers, and unbounded loops are difficult to access for fuzzers with random-based mutations [9–12]. Therefore, recent approaches aim to obtain additional information about the SUT and use it for the test generation. Cha et al. use static analysis to optimize the mutation rate based on the dependencies between multiple bytes [13]. While Cha et al. employ static analysis to identify dependencies on a single execution path, our approach considers all execution paths. Moreover, they are optimizing the mutation rate to target a certain program path, whereas we are employing constraint solving to target one execution path and, thus, are able to specifically execute only this program paths and no one else. Using Driller, a hybrid vulnerability discovery tool, Stephens et al. combine fuzzing and the so-called "concolic execution," a combination of "concrete" and "symbolic execution," to identify magic bytes that cannot be found efficiently with random mutation due to the large input space [11]. We rely on static analysis that, in addition to the magic bytes, also provides us with information on suspected vulnerabilities which we target in our subsequent fuzzing campaign. Corina et al. use static analysis to identify data structures of kernel drivers [14]. This involves creating valid data structures whose fields are filled with randomly generated data. This data may pass the input data validation and, thus, reach deeper code. While we are also using static analysis to identify relevant data structures, we do not fill them with random data but employ constraint solving as described above. Rawat et al. have also addressed this problem with their tool VUzzer that uses static and dynamic analysis to extract properties of an application's data and control flow [9]. It uses this information to prioritize promising paths during test execution without having to use elaborate symbolic execution such as Driller, thus realizing directed fuzzing. In particular, it

[3] A collection of vulnerabilities discovered by the AFL fuzzer can be found under https://github.com/mrash/afl-cve

aims to avoid error-handling code, considering it to contain only a few bugs [9, 15]. In contrast to Rawat et al., we do not target those paths that were executed rarely but those who are considered to contain a suspected vulnerability identified by the static analysis. Böhme et al. pursue a similar approach with AFLGo and use simulated annealing to prioritize between fuzzing as done by AFL and targeting specific code fragments [16]. They identify potential uses as regression testing of patches; reproduction of crashes that occur in the field and are otherwise difficult to reproduce due to missing data, for identifying potentially dangerous dataflows; and verification of static analysis results. In their paper, Böhme et al. also discuss the challenges that symbolic execution faces in the context of patch testing on the example of the Heartbleed Bug. The authors claim that directed fuzzing is superior to symbolic execution since it is too expensive to analyze all possible paths to cover the entire code. This is an actual drawback of symbolic execution when dynamic analysis tries to cover all the code from a patch and, thus, is, for instance, not able to detect the Heartbleed Bug within 24 hours [16]. Even though this may be a problem when applying symbolic execution only, we apply constraint solving only on those program paths that have been previously identified by the static analysis and, thus, overcome this drawback. Furthermore, if the constraint solving fails to state if a constraint problem for a certain program path is satisfiable, we are employing directed fuzzing with the tool AFLGo to target the code in question. Pham et al. apply an approach with AFLSmart that does not only optimize the lines of code to be covered but likewise draws on information about the input data format using a parser to apply mutation operators at the data structure level rather than at the bit level (as AFL does) to leverage knowledge about the application [17]. AFLSmart does not take into account information from static analysis as we do in our approach.

Further, fuzzing approaches combine fuzzing and symbolic execution. One approach uses this combination together with bounded model checking to identify vulnerabilities in C programs [18]. To do so, security properties are verified using model checking to identify if a specific execution path of a C program might violate them. However, as the authors describe, their approach has limitations if large amounts of complex data need to be initialized at the beginning of the program. Complex data is, for instance, required to trigger vulnerabilities such as the Heartbleed Bug. As we demonstrate in the subsequent sections, this is particular where our approach is successful. Another combination of fuzzing and concolic execution has been investigated by Borzacchiello et al. [19]. They derived symbolic queries from a binary using virtualization and applied fuzzing techniques to solve these symbolic queries. However, Borzacchiello et al. do not apply fuzzing techniques to the SUT itself while aiming at better code coverage and performance.

Ognawala et al. use fuzzing together with targeted symbolic execution in a two-step approach [20]: First, they aim at identifying bugs in the SUT by fuzzing isolated functions. Next, their framework analyzes for each vulnerability if it is reachable when considering the call graph through the function call chain until the main function of the SUT is reached. Thus, vulnerabilities that are unreachable by an attacker are identified, since their inputs are sanitized by other functions in the call graph. The main difference between the approach by Ognawala et al. and ours is that we are applying static analysis to identify suspected vulnerabilities and to improve fuzzing to reach parts of the code that is hard to execute by gray box fuzzing such as AFL.

One major issue when applying dynamic analysis is that proving the absence of bugs is not possible. One approach to cope with this issue are statistical methods that can be applied to estimate the residual risk that a bug is not yet discovered but might exist. These methods have been established in gray box fuzzing, and one established statistical method is the Good-Turing-Estimator (GTE) [21]. The basic idea of the GTE is to estimate the likelihood to discover an unseen species based on how often new species have been discovered in the past. This approach has been applied to gray box fuzzing by considering discovered execution paths as species and, thus, mapping the problem of species discovery to software testing [22]. The approach has been improved by Böhme et al. by including an adaptive bias that aims at taking the increasing likelihood to discover bugs into account while the fuzzing campaign advances and code coverage increases [23]. In our approach, we apply this approach to discovering a particular execution path.

### *6.2.1 Interactive Application Security Testing*

While there is a plethora of publications on the techniques that are employed for IAST, e.g., static analysis, constraint solving, and fuzzing, this is not the case for IAST itself. Earlier literature focuses on how to improve automated test case generation using static analysis or automatically generate test cases to verify static analysis findings. For example, Bozga et al. propose to reduce the state space explosion in test case generation for conformance testing through slicing and, thus, simplifying the specification in the form of extended state machines by identifying those parts that are relevant for conformance testing, using a set of given seeds, in contrast to our approach that focuses specifically on vulnerabilities than on conformance testing [24]. Chebaro et al. employ in addition program slicing to reduce the overhead of test execution by removing irrelevant instructions from the code with respect to a static analysis finding identified using value analysis [25]. Their approach results in three states for each finding, i.e., alarm when a finding is a true positive when at least one test case identified a bug at runtime, safe when it is a false positive because all execution paths have been executed and none triggered the bug, and unknown when no test cases triggered a bug but not all paths could be covered. Hence, the approach does not provide any statistical guarantees for false positives for real-world programs, as our approach does. Wang et al. propose a static method to identify Integer errors in the source code using constraint solving based on the source and, thus, enables the identification of vulnerabilities resulting from them [26]. In a similar way, Liang et al. employ symbolic execution and constraint solving to identify vulnerabilities in the source code that result from division by zero error, pointer overflows, and dead code [27]. However, they are suffering from false positives. In contrast, our approach allows the identification of false positives through dynamic analysis and compared to Wang et al. without user intervention.

Williams et al. aim at maximizing path coverage by instrumenting the code of an SUT to collect constraints when executing a single test case and iteratively change their evaluation and use constraint solving to generate a test case that covers a new path in each iteration, thus maximizing the code coverage [28]. We do in addition employ static analysis results to obtain the location of a vulnerability in the code and execute specifically this code (more precisely, the corresponding program path) to discover vulnerabilities. Hybrid fuzzing approaches, such as white box fuzzing, work similarly. Godefroid et al. derive fuzz test data from the source code by collecting all the constraints that were involved when processing a certain valid input [29, 30]. The collected constraints are then negated or otherwise violated, and subsequently, these mutated constraints are used to generate test data. Thus, they achieve a high code coverage but do not employ static analysis to identify suspected vulnerabilities and cannot provide information on the absence of vulnerabilities as our approach aims to do.

## 6.3 Methodology

This section is intended to illustrate the principle methodology to combine static and dynamic analysis. After a brief overview of the information that can be obtained from each analysis method and the variations in IAST approaches, the rest of this section discusses how SAST and DAST can support and improve each other. Both static and dynamic analyses consider the SUT on different stages. While static analysis examines the source code, dynamic analysis investigates the runtime behavior. Each analysis technique can obtain quite different types of information about the SUT.

Static analysis identifies potentially vulnerable program code by searching for dangerous patterns such as the unsafe usage of user input. To do so, all program paths are analyzed. By identifying such patterns, static analysis can provide concrete information about the potential vulnerability, e.g., its type and its location. In addition, it can be analyzed under which conditions the vulnerable code is reachable by collecting the conditions of the related program path. However, due to the "exponential explosion" of potential program state configurations and the large input space, a complete analysis of all feasible configurations and inputs is usually not possible.Therefore, abstractions are necessary to be able to perform the analysis which leads to the fact that it also identifies a large number of potential vulnerabilities that may not occur during execution because the conditions that lead to the vulnerability cannot be satisfied, called false positives.

In contrast, dynamic analysis observes the system at runtime to determine properties that hold for one or more execution [31] and can thus gain information about the system that cannot be determined statically. The explicit execution of the SUT ensures that false positives are rarely generated since vulnerabilities are only reported only if they are observed during execution. However, through dynamic analysis, it is often not possible to make a precise statement about the

**Table 6.1** Strengths and weaknesses of static analysis and dynamic analysis

| | Static analysis | Dynamic analysis |
|---|---|---|
| Strengths | - High path coverage<br>- Good presentation of results | - Few false positives<br>- Provides input triggering a vulnerability |
| Weaknesses | - High number of false positives | - Random path coverage<br>- Poor result presentation<br>- Less code inferences |

location of the vulnerability in the source code. In contrast, dynamic analysis can provide the input which caused the vulnerability to be executed and, thus, support reproducing and investigating the vulnerability during debugging. When performing dynamic analysis for security testing, a common goal is to maximize the code coverage and, thereby, execute as much potentially vulnerable code as possible. Without knowledge about the implementation, this can be a tedious task as some parts of the code may only be reached under complex conditions that are rarely satisfied. Therefore, with purely no knowledge about the implementation, only random path coverage can be achieved. Here, static analysis can provide the missing knowledge to increase the code coverage much faster. In summary, both methods have complementary strength and weaknesses, which are summarized in Table 6.1.

### 6.3.1 *Interactive Application Security Testing: Combining Static Analysis and Security Testing*

When both analysis methods are used separately, they are limited to the information they can obtain. Therefore, interactive application security testing (IAST) combines static and dynamic analysis to benefit from both methods. Forwarding information from static to dynamic analysis (cf. Sect. 6.3.2.3) and vice versa (cf. Sect. 6.3.2.2) can significantly improve the effectiveness of both methods. Moreover, they can compensate for each other's weaknesses, resulting in significant gains in efficiency and accuracy compared to their independent application, in terms of vulnerability discovery with respect to true and false positives and negatives, efficiency, and manual effort. However, the term "interactive" in IAST does not specify when both approaches interact. The exchange of information is furthermore not limited to a single exchange. New analysis results and information can be continuously exchanged, such that both analysis methods can repeatedly receive new information and improve their analysis, which in turn can lead to further information for the respective other analysis approach. Different interaction approaches are possible, and which suits best depends on the specific goals. Basically, we identified three

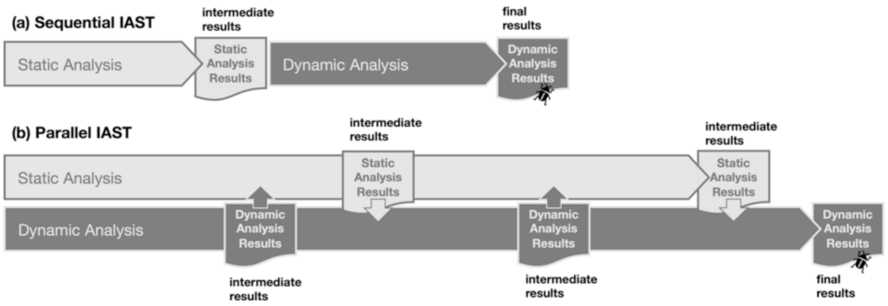


**Fig. 6.1** Sequential and parallel IAST approach

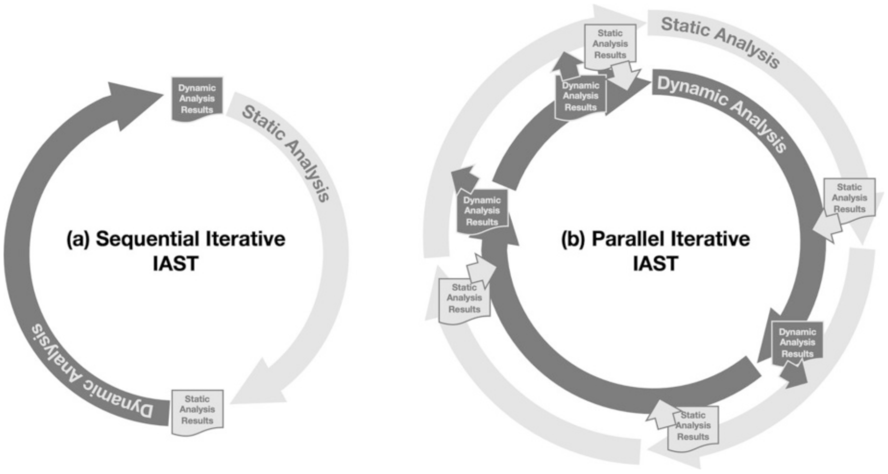


**Fig. 6.2** Iterative IAST approaches

different interaction models (see Figs. 6.1 and 6.2), which can be applied in different variations:

In the *sequential IAST approach*, depicted in Fig. 6.1a, static and dynamic analyses run one after the other. As soon as one analysis, e.g., static analysis, has finished, its information is passed on to the other, which in turn uses this information to perform and improve its own analysis. This approach is useful if the results from the static analysis should be verified through dynamic analysis. When both analyses have been run, the execution is finished. It runs first the static analysis with a single run that provides intermediate results (from the IAST point of view) to the dynamic analysis, which in turn uses this information to perform its own analysis. It finishes with the provision of the final analysis results, i.e., the identified vulnerabilities. We expect the dynamic analysis usually to provide final results since it is used to verify the findings from the static analysis (cf. Sect. 6.3.2).

The *parallel IAST approach* illustrated in Fig. 6.1b is similar to the iterative approach. However, it exploits the fact that both analyses can run on their own, i.e., dynamic analysis starts at the same time as the static analysis. Since both analyses can run independently, vulnerabilities can be earlier identified, and information valuable for the respective other analysis can be obtained earlier than in the sequential approach. Thus, both analyses can benefit from each other. When they receive results from the other analysis, they can take them into account to improve their analysis. Both static and dynamic analyses start independently from each other. Here, the dynamic analysis would provide intermediate results to the static analysis, e.g., values of local variables. The static analysis can use this information to perform a more accurate static analysis. When it has calculated first results, the dynamic analysis can use the information from the static analysis, e.g., magic bytes, to increase code coverage more efficiently, for example. Both analyses can exchange information until they have completed their run.

The *iterative IAST approach* extends the sequential or parallel approach to a cycle in which both are executed repeatedly (see Fig. 6.2). The iterative approach enables to propagate results in both directions such that static and dynamic analyses benefit from each other where new analyses could be run based on further information. Moreover, the analysis can run as long as new results can be identified or previous results could be refined. For example, the static analysis could be more precise because of concrete values it receives from the dynamic analysis, such as addresses of pointers. Figure 6.2 depicts the sequential (a) and parallel iterative approach (b). In the sequential iterative approach, static analysis propagates its results to dynamic analysis. As in the sequential approach, the dynamic analysis uses these results for its own analysis. In contrast to this, the iterative approach continues when the dynamic analysis has been finished, allowing the static analysis to benefit from the dynamic analysis, similar to the parallel approach. The parallel iterative approach combines the benefits of both the sequential and the parallel approaches, allowing them to start independently and propagate results as soon as they are available. In addition, it allows to repeat and refine the analyses when completed to improve its accuracy, reducing false positives and execute further program paths.

## *6.3.2 Our Approach to IAST*

For our approach of IAST, we identified three main goals which are:

1. Verify SAST findings using DAST.
2. Improve DAST with SAST results.
3. Improve SAST with DAST results.

It seems obvious that for the first goal, static analysis should start before the dynamic analysis and after static analysis has identified candidates for vulnerabilities, dynamic analysis is run to discriminate the true and false positives, a tedious task that is expensive if done manually. Toward this goal, we can use a sequential IAST approach since the dynamic analysis requires the results from the static analysis to start. Hence, dynamic analysis can run as soon as the static analysis provides its first finding. In addition to pure static analysis, dynamic analysis can handle situations in which the static analysis would fail, e.g., if the constraint solving does not terminate (cf. Sect. 6.3.2.1) and provides a test case that triggers the vulnerability and, thereby, supports the patch development.

The second goal is to improve dynamic analysis with static analysis results. We could achieve this by exploiting magic bytes to increase code coverage more efficiently. Toward this goal, dynamic analysis needs to wait for such information. However, magic bytes, for instance, are not needed from the beginning of the dynamic analysis but can help to improve code coverage at a later point in time. Hence, it is useful to run static and dynamic analysis in parallel.

Toward the third goal, static analysis may benefit from dynamic analysis results by obtaining information it has to abstract from. Such information can help to make the static analysis more precise by augmenting it with information it cannot calculate on its own and, thereby, addressing one of its drawbacks partially. For this purpose, the dynamic analysis needs to run first before the static analysis can retrieve such information. This is similar to the second task with reversed roles of static and dynamic analysis. Both models can be integrated where both static and dynamic analyses run in parallel and forward the information to the respective other analysis. In the current stage, we are applying the parallel IAST approach. However, all the experiments described in Sect. 6.5 can be conducted using the sequential IAST approach.

#### 6.3.2.1 Dynamic Verification of Static Analysis Findings

To verify the findings of the static analysis dynamically means to determine whether a finding is a true or false positive. For this purpose, specific test cases are generated, which are intended to verify the analysis findings, i.e., execute the suspected vulnerabilities. If the suspected vulnerability can be observed, a finding is confirmed to be a true positive. Since *"testing can be used to show the presence of bugs, but never to show their absence"* [32], as long as the vulnerability cannot be observed, in general, no reliable statement can be made whether a finding is a false positive or not (even though, in certain cases, false positives can be identified, cf. Fig. 6.3). Therefore, a test suite is needed that provides some evidence that the finding can be considered as a false positive with some certainty. Here, statistical methods can be used to calculate a residual risk, which is a measure of the probability that an analysis finding could be a true positive despite the fact that it is not triggered by the test campaign that has been executed so far. In the following, the proposed process for the generation of test cases, the identification of true and false positives, and the calculation of the residual risk will be discussed in more details.

To be able to test a SUT for a certain vulnerability (reported by the static analysis), three artifacts are relevant that constitute a test case: abstract test case, test data, and a suitable test oracle.

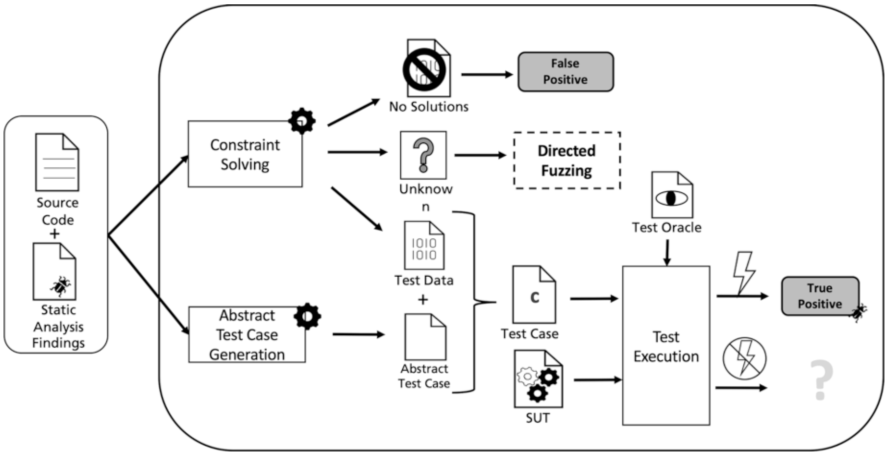


**Fig. 6.3** Verifying static analysis findings using constraint solving

Abstract Test Case The abstract test case specifies the entry point and the sequence of test steps without concrete test data. It describes if a single function of the SUT (e.g., a library), a combination thereof, or a certain interface is tested.

Test Data Test data are the concrete inputs used by a test case and affect the execution of the SUT. Through suitable test data, it can be controlled which part of the function invoked by the abstract test case is executed. In addition, vulnerabilities are often only triggered with suitable test data, e.g., buffer overflows. Hence, covering the statement containing the vulnerability is not always sufficient.

Test Oracle Just as important as the use of suitable test data, the choice of the right test oracle is crucial to be able to observe a vulnerability appropriately, because not every bug or security issue can be observed through a crash of the SUT.

It is important that the static analysis provides precise information on the location of a suspected vulnerability, i.e., in which function and line of code it is located. The location of the vulnerability is relevant for two reasons. On the one hand, the location information specifies which target region of the code the test case shall exercise. The test data must therefore be generated such that it executes a program path that leads to the execution of the provided target region. On the other hand, the vulnerabilities observed during test execution need be compared with the vulnerability provided by the static analysis in terms of its location and its type to verify if a vulnerability provided by the static analysis has been confirmed or if an additional has been discovered.

Once the static analysis provides the location of a potential vulnerability, test cases can be generated that aim to execute the given part of code. In addition to the information on the location, the identified call chain that leads to the suspected vulnerable line of code may be useful. In some cases, it can be sufficient to use only the function enclosing the vulnerable code in the abstract test case. However, in some cases, the previous call of other functions of the call chain is required since they lead to a state of the system that causes the vulnerability to be triggered. Therefore, it is reasonable to create not only a test case that executes the possibly vulnerable line of code but to create a number of test cases to increase the probability of discovering the vulnerability.

Furthermore, the information on the type of the vulnerability that the static analysis provides is crucial in addition to the location information. The type of the vulnerability is relevant to select and conFig. the test oracle appropriately. Only if the type of the vulnerability is known a suitable test oracle can be selected to observe the vulnerability during test execution. In addition, it should be considered that not only the suspected vulnerability may occur but also additional ones that are located on the very same execution path. Thereby, it is a great benefit if the test oracle is also able to provide information about the responsible source code (location information) when a vulnerability is detected. This information is needed for comparison with the prediction of the static analysis. If the information matches, a finding can be classified as a true positive; otherwise, an additional vulnerability has been found.

To reach a certain target region with a test case, the test data must be selected such that the desired program path is executed, which means that branches and jumps are selected accordingly. However, for some types of vulnerabilities, such as the double-free vulnerability, it is not enough to execute only the corresponding line of code; instead, certain instructions have to be executed beforehand. This information can also be provided by the static analysis. Accordingly, the target region can be considered as a set of code instructions that must be executed during runtime. To suffice the aforementioned requirements, test data can be determined by using constraint solving. In the case of constraint solving, the program statements of the respective program path will be translated into formal logic expressions and collected in a so-called constraint system. Note that we need to consider only that part of the program path from the beginning of the function to the vulnerable line of code and can ignore the remainder of this path.

To solve a constraint system, a so-called constraint solver is employed. By solving the constraint system, the input values for the abstract test case are obtained, which would cover the desired target region during test execution. In addition to the translated program statements, vulnerability constraints can be added to the constraint system. The vulnerability constraints specify the conditions that must be satisfied to trigger the vulnerability. For instance, in the case of an overflow vulnerability, this could be a constraint that a value should be larger than the given buffer. When trying to solve the constraint systems, the following three situations can occur:

1. The constraint system is satisfiable.
2. The constraint system is not satisfiable.
3. The constraint solver is not able to solve the constraint system.

In the case that the constraints system is satisfiable, the constraint solver can provide one or more solutions for the constraint system. Each solution can be used to generate test data that, together with the abstract test case, form a test case that aims to trigger the vulnerability. The generated test case is then executed against the SUT. If the suspected vulnerability can be observed by the test oracle, a true positive is clearly identified. If not, no clear statement can be made about the existence of the vulnerability in question. In that case further test cases need to be executed. If the constraint system is not satisfiable, the corresponding program path is unfeasible, that is, its instructions cannot be executed while at the same time satisfy all its constraints and from the vulnerability type. If this applies to all possible program paths of a suspected vulnerability to be checked, it can be stated that this finding from the static analysis is a false positive since there is no test data that can lead to the suspected vulnerable line of code. The described process is illustrated in Fig. 6.3. The individual steps are fully automated. Only the configurations required for test execution, such as the compiler to be used or header files to be included, must be configured manually beforehand.

In some cases, it may happen that the constraint solver does not terminate or cancels the process without providing an appropriate answer whether the constraint system is satisfiable or not. In such a case, we cannot rely on constraint solving for the test data generation. If none of the other program paths of the resulting constraints system already lead to executable test cases that might trigger the suspected vulnerability, a fallback solution is required. Here, directed fuzzing can be applied to verify the static analysis finding. Directed fuzzing is a gray box fuzzing technique which aims at covering a certain target region of the SUT's code instead of covering its entire code. In directed fuzzing, the SUT is instrumented to give feedback on how close an execution has come to the target region. Based on that feedback, certain test inputs are selected and further modified in an attempt to get closer and closer to the desired target region.

#### Residual Risk Estimation

To estimate the residual risk that a vulnerability that has not been detected so far nevertheless exists, estimation heuristics can be used. One of such estimation heuristics that has been applied in the context of fuzzing is the GTE [21]. The GTE can be used to estimate how likely it is that the next sample of an observation is a previously unseen element. Applied to fuzzing, the GTE can estimate that the next test case executes a previously unobserved execution path. The estimator has been used in the context of gray box fuzzing by Böhme et al. [22]. Their experiments have shown that the estimator provides a reasonable upper bound for the success of a fuzzing campaign. The necessary calculation is relatively lightweight and the required amount of data relatively small. Stated by Good and Turing [21], the probability $P_0$ that the next execution path is a path that has not been observed before is approximated by the number of test cases that thereby produced a unique execution path divided by the total number of test cases that have been executed so far:

$$P_0 \approx \frac{\text{\# unique execution path}}{\text{\# test executions}} \tag{6.1}$$

In contrast to Böhme et al. [22] who used the GTE to estimate whether further bugs can be detected when continuing a fuzzing campaign, we use the GTE to estimate the residual risk for a specific vulnerability to assess whether it is a false positive. The GTE can be calibrated such that the test campaign is stopped when the calculated estimation falls below a certain threshold. Here, looking at a single, absolute GTE value is not sufficient, and taking into account how the value has evolved over the course of the previous test executions is required. When the test execution is stopped, the residual risk that the suspected vulnerability is not a false positive even though it has not been observed during test execution can be reported.

#### 6.3.2.2 Improving DAST with SAST Results

One major advantage of IAST is that both analyses can exchange information to augment the respective other analysis with information it cannot obtain itself but would make the respective analysis more accurate or faster. As mentioned above, most DAST tools suffer from the lack of knowledge of the internals of the SUT and can therefore only perform a more or less random-based dynamic analysis. Using information supplied by the static analysis, more targeted test cases can be generated and executed to reach the desired code more quickly, e.g., increase the code coverage or reach deeply nested program paths. Here, static analysis can provide information on the requirements an input must meet to execute a certain part of the code. Therefore, the static analysis can provide information on the related path conditions, for instance, that an integer must be set to a specific constant value or that a string must contain a specific character. We call these conditions "magic bytes." Magic bytes can be used to support fuzzing because they can tell us which inputs need to be fixed to what values so that we reach a certain location, e.g., "$x == 10$." A random-based approach is not able to take this information into account. We can then focus on producing random values only for the unconstrained inputs while keeping the constrained ones fixed. Additionally, this would arguably allow for a more low-cost exploration of a target region than input generation via constraint solving.

**Table 6.2** Information exchange between SAST and DAST

| SAST → DAST | DAST → SAST |
|---|---|
| - Vulnerability type | - Runtime information, such as memory references |
| - Vulnerability location | - Values of local variables |
| - Identified pattern | - Results of external functions |
| - Path constraints | - Appearing side effects |
| - Magic bytes | |

However, static analysis cannot only contribute to faster code coverage. It can also provide information about certain insights it gathered, e.g., third-party libraries and the used programming languages. Such information can be used to generate more targeted test cases. For instance, if it is known that certain inputs are used for database queries, these inputs can be specifically tested for injection vulnerabilities.

#### 6.3.2.3 Improving SAST with DAST Results

On the other hand, static analysis can benefit from dynamic as well. By its nature, static analysis cannot precisely identify information such as references and assignments that are performed dynamically or the resource consumption at runtime. This is where dynamic analysis comes into play. The dynamic analysis observes the SUT at runtime to determine properties that hold for one or more executions [31] and can thus gain information about the system that cannot be determined statically, e.g., the function a function pointer is referring to at a certain stage. For this purpose, the SUT is usually executed through test cases aiming to execute the related code of the SUT. Table 6.2 provides an overview on the information one analysis can pass over to the other.

## 6.4 Implementation

In the course of this section, we present mainly the realization of the presented methodology for generating test cases that are used to verify the static analysis findings. The Heartbleed Bug serves as a running example for demonstrating the implementation details. The Heartbleed Bug[4] is a buffer overread vulnerability in the heartbeat protocol implementation of the OpenSSL library, which was introduced in 2012 to enable a low-cost, keep-alive mechanism between client and server. In the heartbeat protocol, the client could pass a payload and its length to the server and receive the same payload back from the server in response.

[4] CVE-2014-0160

```
 1 #define TLS1_HB_REQUEST 1
 2 ...
 3 int tls1_process_heartbeat(SSL *s) {
 4    unsigned char *p = &s->s3->rrec.data[0], *pl;
 5    unsigned short hbtype;
 6    unsigned int payload, padding = 16;
 7
 8    /* Read type and payload length first */
 9    hbtype = *p++;
10    n2s(p, payload);
11    pl = p;
12    ...
13    if (hbtype == TLS1_HB_REQUEST) {
14      unsigned char *buffer, *bp;
15      ...
16      buffer = OPENSSL_malloc(1 + 2 + payload + padding);
17      bp = buffer;
18
19      /* Enter response type, length and copy payload */
20      *bp++ = TLS1_HB_RESPONSE;
21      s2n(payload, bp);
22      /* vulnerable line */ memcpy(bp, pl, payload);
23      ...
24    }
25    ...
26 }
```

**Listing 6.1** Heartbleed Bug code snippet

However, in the code it was not checked if the passed payload length matches the payload. Therefore, it was possible to specify the payload length to be much larger than the payload itself. This had the consequence that the server read out and returned more data from the memory than it was supposed to. Listing 6.1 shows a snippet of the corresponding source code of the *tls1_process_heartbeat* function,[5] which contains the described vulnerability in line 22.

## 6.4.1 Static Analysis

For the static analysis, the open-source framework PhASAR [33] has been used. PhASAR[6] is an LLVM[7]-based static analysis framework that offers the possibility to specify arbitrary dataflow problems which are then solved in a fully automated way by a so-called dataflow solver. PhASAR provides, among others, several algorithms for analyzing the dataflow. The PhASAR-based static analysis provides several

[5] The snippet is taken from the official OpenSSL GitHub repository https://github.com/openssl/openssl/blob/OpenSSL_1_0_1f/ssl/t1_lib.c.

[6] https://github.com/secure-software-engineering/phasar

[7] https://llvm.org/

pieces of information about a presumed vulnerability. First, its location (specified by the file name, the function name, as well as the line and column number) as well as the type of the vulnerability. In addition, information about the value assignments (*magic bytes*) that must be met and program statements to cover to trigger the potential vulnerability will be provided. In the case of the Heartbleed Bug, the static analysis will report a potential buffer overread vulnerability in line 22 and additionally provide the information that are printed in green and underlined in Listing 6.1. For the interaction between the static and the dynamic analysis, a parallel approach is used. As soon as a vulnerability is suspected, the information is made available to start the dynamic verification whereby the static analysis continues in parallel.

### *6.4.2 Test Case Generation*

For each reported vulnerability, the function in which the vulnerability is located (provided by the static analysis) is extracted and called by the generated (abstract) test cases.[8]

To determine the possible execution paths that meet the requirements described above, a State Machine (SM) representation of the source code is generated. Each basic block of the source code is translated to an action, and branch conditions are used as guards of the transitions between its states. The states have no actions associated with them. The information provided by the static analysis about the program statements to be executed (including the line with the potential vulnerability) are mapped to the corresponding transitions of the SM that need to be covered. For the test case generation, all possible paths starting from the initial state to the presumed vulnerable line of code (reflected by the corresponding transitions of the SM) are determined, taking into account the transitions that need to be covered. Here, the execution paths are only considered symbolically, which means that only the abstract paths are considered but no concrete values are used such that the guards of the corresponding transitions can be ignored at this point. Figure 6.4 shows the SM representation of the function *tls1_process_heartbeat* from Listing 6.1. The transition marked in red includes the vulnerable line of code (line 22 in Listing 6.1) that needs to be executed by the test case. Consequently, all paths containing the red marked transition need to be considered for the test case generation.

[8] Currently, only the specified function is tested. In the later course of the project, the calling functions will also be taken into account to decide whether a vulnerability can be exploited or not.

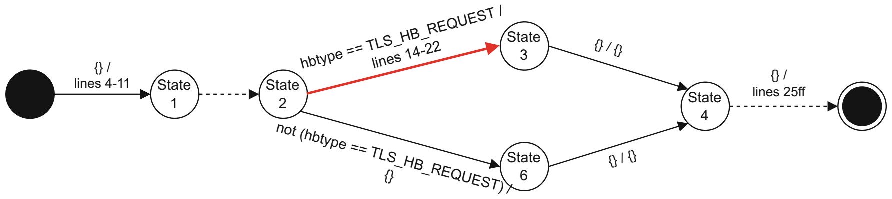


**Fig. 6.4** State machine representation of function *tls1_process_heartbeat*

### 6.4.3 Test Data Generation

For the test data generation using constraint solving, the transformation of the source code into the SM translates the individual program statements into constraint expressions (actions and guards from the transitions) that can be directly used by the constraint solver. The particular constraints of each transition on a path are collected along the path. To create the constraint system, the targeted path is traversed, and for each transition, the constraint expressions are added to the constraint system.

We have chosen the Z3 Satisfiability Modulo Theories (SMT) solver [34] as our constraint solver. Therefore, each program instruction is translated into a Z3 constraint expression. These expressions are grouped by the transition they belong to. In addition, at each *branch transition* (i.e., each transition originating from a state with more than one successor), we can generate *guard expressions*: Boolean expressions which must evaluate to *true* or *false* if one of their associated transitions should be taken. Z3 already offers a large number of functions to translate basic expressions and operations (like arithmetic operations) into Z3 constraints.[9] Primitive data types are represented as bit vectors. For complex data types as well as for arrays, Z3 offers to create the so-called data types and array expressions.

To use *data types* and *array expressions*, however, the data types and array lengths must be known, which is not necessarily the case when pointers are used by the SUT's code. Pointers are a heavily used feature in programming languages like C. However, there is no obvious method of representing pointers in Z3. Pointer handling is thus by far the most complex part of constraint generation. Pointers are modeled as a data-type expression that consists of a buffer ID and the indices used by `getelementptr` (which identifies an element of the buffer). A `PointerManager` keeps track of the buffers corresponding to the IDs and updates them on `store` instructions. The buffer elements are typed according to their types in the LLVM IR, allowing us to use Z3's rich-type system (*Sort* in Z3 terminology). This is only possible because of LLVM's typed pointers. However, LLVM is in the process of eliminating pointer types and transitioning to opaque

[9] We use here the Z3 Java bindings provided by Z3 itself.

pointers.[10] To add the vulnerability constraints, as described in Sect. 6.3, we first identify the program statement by which this presumption was made and which variables have an influence on the triggering of the vulnerability. For example, in the case of a suspected buffer overread vulnerability which is associated with a *memcpy* function call,[11] a constraint would be added to the constraint system that the number of characters that should be copied (third function argument) must be larger than the length of the given buffer (second function argument).

In the case the constraint solver (here Z3) can provide a solution, this solution will be transferred into concrete test data by the test data generator (see Fig. 6.3) that translates the provided solution into the test case language (e.g., C). This means not only the assignment of variable values but also the creation of data structures and pointers are expected as function arguments for the function under test. If there is no solution – meaning the constraint system is not satisfiable – the execution path is marked as "not satisfiable," and the remaining paths are checked. If all possible paths are marked as "not satisfiable," the suspected vulnerability is declared as a false positive.

In case that the Z3 constraint solver is not able to provide a solution (for at least one path), we choose AFLGo [35] to perform the directed fuzzing. AFLGo is an extension of the open-source fuzzing tool American Fuzzy Lop (AFL). Unlike pure AFL, it is designed to target specific code locations in the SUT, making it especially useful for the verification of static analysis reports. AFLGo extends AFL's instrumentation that merely utilizes branch coverage to also include path distance information. Path distance information is calculated during the compilation of the SUT so that there is no performance loss at runtime. Path distance information is then used during the fuzzing campaign to choose those seeds that specifically lead to targeted regions in the SUT, while other seeds that, even if revealing new paths, lead further away from these regions are willingly omitted. The fuzzing algorithm of AFLGo is basically the same as that of its ancestor AFL. It starts by feeding user-supplied test cases into the instrumented SUT, repeatably mutating them slightly, utilizing a variety of traditional fuzzing strategies, to produce inputs that trigger new state transitions. If such a new transition is recorded, the responsible input is added to a queue of seeds that then undergoes the same procedure. To feed the test data to the SUT, there has to be an interface between AFLGo and the SUT. Here, a test adapter is needed that takes the test data generated by the fuzzer and wraps it in a way that the SUT can handle it (e.g., fill some complex data types).

[10] In LLVM-14, typed pointers have been deprecated, and they will be removed in LLVM-15 https://llvm.org/docs/OpaquePointers.html.

[11] C library function voids that copies n characters from one memory area to another.

### *6.4.4 Test Oracle*

The test oracle employed is selected based on the suspected type of vulnerability. Currently, the implementation is focused on *buffer overread* and *double-free* vulnerabilities. Both can be observed with the *AddressSanitizer* (ASAN). *"The AddressSanitizer is a fast memory error detector that consists of a compiler instrumentation module and a runtime library"*.[12] It can be used by simply compiling and linking an application with a certain flag. If there is a *buffer overread*[13] or *double-free* vulnerability observed at runtime (triggered by the test case), this is reported by ASAN. In addition to ASAN, there are other sanitizers like *MemorySanitizer*[14] that will also be included in the future.

The report of the test oracle (here ASAN) is then compared with the information about the vulnerability provided by the static analysis. Does the provided information match with what has been determined at runtime, the suspected vulnerability is a true positive. If not, it can be declared neither true nor false positive, even if the ASAN has reported a vulnerability (then probably an additional vulnerability has been discovered).

Since it cannot be stated that there is no vulnerability in the case that the vulnerability in question was not detected, new test data must be generated for further test cases obtained from the constraint solver, which applies to the directed fuzzing approach as well. To estimate the residual risk for each test execution, the executed program paths must be extracted. This specific coverage information is then processed and used by the Good-Turing estimator (GTE) (see Sect. 6.3.2.1) to assess the progress of the test campaign.

## 6.5 Evaluation

To answer the research questions as well as to verify the established hypotheses, different experiments were carried out which should provide the corresponding insights. For the evaluation, the OpenSSL library in version 1.0.1f has been selected as the SUT, which contains the Heartbleed Bug described in Sect. 6.4. To provide a baseline approach for the evaluation, the experiments with the OpenSSL library using the presented approach will be compared to an undirected gray box fuzzing approach using AFL[15] and its directed fuzzing counterpart AFLGo.[16]

[12] https://clang.llvm.org/docs/AddressSanitizer.html

[13] More general out-of-bound access

[14] https://clang.llvm.org/docs/MemorySanitizer.html

[15] https://github.com/google/AFL

[16] https://github.com/aflgo/aflgo

### *6.5.1 Experimental Plan*

The goal of the experiments was to compare the performance of the different approaches, i.e., undirected fuzzing, directed fuzzing, and our IAST approach, by means of their implementations, in terms of runtime and required test cases to detect the Heartbleed Bug. For all three approaches, the runtime from the start of the fuzzing campaign until the detection of the Heartbleed Bug, the time to exposure (TTE), has been measured. This includes the test case generation, the test case execution, and the test evaluation. The time needed for the compilation of the tools, instrumenting the SUT and, in case of directed fuzzing, for calculating the distances to the target region have not been considered. Also, the runtime of the static analysis tool has not been included since directed fuzzing and our IAST approach are independent from a specific tool, and hence, every static analysis tool could be employed.

In addition to evaluating the runtime and the number of test cases required to trigger and detect the Heartbleed Bug, the aim was to investigate how meaningful the used GTE is for estimating the residual risk that a vulnerability will be discovered in future test executions. To do so, the GTE values (see Eq. 6.1, Sect. 6.3.2.1) have been calculated during the abovementioned experiments. After the execution of each test case, the number of newly discovered, unique execution paths is calculated and divided by the number of total test executions. All the experiments for the evaluation have been conducted on a machine running Ubuntu 20.04.4 LTS on an Intel Xeon E5-2680 v4 processor with 8 cores running at 2.4 GHz, 32 GB RAM, and 72 GB HDD memory. The respective tools, namely, the static analysis, AFL, ALGo, as well as our IAST approach presented in Sect. 6.4, were running in separated Docker containers on this machine.

### *6.5.2 RQ1: Information Exchange*

*Which information can static analysis provide to DAST that facilitates its analysis?*

Static analysis can provide information about the code region where potential vulnerabilities may be located to guide the dynamic analysis to this code region. Even though this may reduce the effort on the dynamic analysis, the experimental results in Table 6.3 clearly show that knowledge of a potential vulnerable region alone is not sufficient for this purpose. In our hypothesis related to RQ1, we assume that static analysis can guide the dynamic analysis where in the code to search for vulnerabilities and can even provide further information that enables targeted testing for the vulnerabilities.

The results from the experiments that have been conducted for this research question are presented in Table 6.3. To avoid that a single run would result in an exceptionally short or long test campaign due to the inherent randomness of the fuzzing approach, each experiment has been repeated 30 times as recommended by

**Table 6.3** Mean ($\mu$) and standard deviation ($\sigma$) of time to exposure (TTE), test cases (TC), and test cases to exposure in the target region (TCTE) for AFL, AFLGo, and Our IAST approach to trigger the Heartbleed Bug

| | TTE (sec) | | #TC | | #TCTE | |
|---|---|---|---|---|---|---|
| Approach/tool | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ |
| Gray box fuzzing/AFL | 854 | 1,132 | 35,221 | 48, 495 | 21 | 46 |
| Directed fuzzing/AFLGo | 5,536 | 2,787 | 14,916 | 7,573 | 4 | 8 |
| Our IAST approach | 23 | 3 | 1 | 0 | 1 | 0 |

[36]. Table 6.3 shows the average TTE and the average number of test cases executed by all the approaches as well as the standard deviation. Additionally, Table 6.3 shows the number of test cases that covers the vulnerable line of code (line 22 in Listing 6.1). The gray box fuzzing approach from AFL serves as our baseline approach and requires about 35,221 test cases that are generated and executed in about 14 minutes to detect the Heartbleed Bug. On average, of the 35,221 test cases, 21 test cases reached the target line, but only 1 triggered the Heartbleed Bug.[17] The directed fuzzing approach, knowing from the static analysis the code region that contains the vulnerability, requires for the same task around 14,916 test cases and 92 minutes. Hence, the information provided by the static analysis does not necessarily lead to a more efficient dynamic analysis. Our IAST approach using the similar information as the directed fuzzing approach generates two test cases of which one is executed in about half a minute to trigger the vulnerability. Note that the time needed for the static analysis is neither included in the directed fuzzing approach nor in our approach.

The reason that AFL and AFLGo perform worse is that gray box approaches craft their test cases in small steps. The evolutionary algorithm performs in each generation small mutations on the test cases from the previous generation and only indirectly exploits the information on the code itself, i.e., through the discovery of new execution paths. This leads to a large number of similar test cases that differ only slightly from each other, requiring many generations until the desired execution path that executes the Heartbleed Bug has been reached. In our IAST approach, we employ in addition to the code region that contains the vulnerability also the path conditions that need to be satisfied to execute the vulnerability. By providing this information, test cases can be generated that target specifically the execution path of the vulnerability in question, and thus, the number of test cases to trigger it is significantly lower. Addressing vulnerabilities within each function independently is especially important for libraries. Their functions must also be considered independently of each other, as they can be used in isolation and in different contexts. In summary, we can conclude that static analysis facilitates the work of DAST not only by providing where a vulnerability might be located but also through which program path it may be executed, including the corresponding path conditions, which together significantly improve its performance.

[17] Since the measurement was stopped when the bug was triggered

### 6.5.3 RQ2: Is IAST Worth the Effort?

*Is IAST more efficient than DAST on its own? Under which conditions is IAST more efficient than DAST?*

Our hypothesis here is that IAST plays its advantage when the SUT is well tested and contains only a few, deeply hidden bugs, where complex path constraints constitute a natural barrier for traditional DAST. We use the results from the previous experiments presented in Table 6.3 to discuss this research question and assess our hypothesis.

As can be clearly seen, the creation of targeted test cases, as our IAST approach does, can uncover the Heartbleed Bug in less test cases than DAST approaches. In case of the Heartbleed Bug, our IAST approach generates two test cases. Listing 6.2 depicts one of these two test cases. The reason why two test cases are generated is related to the structure of the function that processes the heartbeat information. The hidden program statements in line 12 of Listing 6.1 contain a conditional functional call, which results in two program paths that both lead to the vulnerable line of code. Hence, two test cases are generated where each one represents one of the program paths: both the test case that covers the conditional function call and the one that does not trigger the Heartbleed Bug. Therefore, practically, only one test case needs to be executed to confirm the bug.

```
void testcase_0(){

      struct ssl_st *ptr0 = ... /* malloc */
      struct ssl3_state_st *ptr1 = ... /* malloc */

      char *ptr2 = ... /* malloc */
      ptr2[0] = ((char) 1); //hbtype
      ptr2[1] = ((char) 0); //payload length
      ptr2[2] = ((char) 4); //payload
      ptr2 += 0;

      struct ssl3_record_st v3 = {...,ptr2,...};
      struct ssl3_state_st v4 = {...,v3,...};
      ptr1[0] = v4;
      ptr1 += 0;

      struct ssl_st v6 = {...,ptr1,...};
      ptr0[0] = v6;
      ptr0 += 0;

      tls1_process_heartbeat(ptr0);
}
```

**Listing 6.2** Generated test case triggering the Heartbleed Bug

To address the second part of the research question, we take a closer look at the information our IAST approach uses from the static analysis. As said in the context of the discussion of RQ1, we use the information from the static analysis where the vulnerability is located as well as the conditions of those program paths that would execute the vulnerability. Although the generation of test cases using constraint solving seems to be more complex, the experiments show that the targeted test case generation as used in our IAST approach that takes into account the additional information about the SUT from the static analysis significantly decreases the number of test cases needed to confirm the bug – which results in a much shorter test execution time – than our baseline approaches that employ evolutionary algorithms used by common DAST approaches like AFL and AFLGo. Both AFL and AFLGo require a large number of test cases to detect the Heartbleed Bug. However, the extent of this advantage depends on the number and the complexity of the path conditions. The more complex these are, the more difficult it is for gray box fuzzing to cover new branches and, thus, increase code coverage of the SUT. A second advantage is related to the number of vulnerabilities in the SUT. Since gray box fuzzing approaches perform a kind of breadth-first search for vulnerabilities due to their aim to maximize the code coverage, they are particularly useful if security testing has not yet been performed and when vulnerabilities may be scattered in large amounts of the code throughout the SUT. In contrast, our IAST approach performs a kind of depth-first search where single program paths are assessed for their feasibility and then executed. Thus, they execute only small pieces of the code which justifies their usage in scenarios where only small parts of the code shall be analyzed or when only a few vulnerabilities are expected in the code, and these maybe located in a certain code region.

In summary, we can conclude that IAST is per se not the best approach in all cases. The advantage of the IAST approach we presented is however relevant when only few vulnerabilities are expected, but it is not known which part of the code accounts for them. Another advantage of IAST can be significant in scenarios where small changes are frequently made to the code, such as in agile development approaches. These can benefit from IAST in comparison to gray box fuzzing since they often add amounts of code and shorter development cycles do not provide much time for comprehensive gray box fuzzing.

### 6.5.4 RQ3 and RQ4: Identifying True and False Positives and Uncertainty

*RQ3: To which extent can false and true positives from static analysis be automatically discriminated by dynamic analysis?*

*RQ4: How well do methods used in DAST to quantify the uncertainty to discover new bugs work in the context of IAST, in particular when we aim at verifying SAST results using DAST?*

Both research questions are closely related, which is why we discuss them together. Related to these research questions is the hypothesis that we can verify true positives from SAST well, but this is much harder for the false positives. This seems obvious since we cannot show the absence of bugs with pure testing, as stated by Dijkstra [32]. However, we aim to extend this a bit toward the identification of false positives using dynamic analysis in combination with a statistical measure to draw better conclusions on the absence of bugs. Statistical estimators such as the Good-Turing estimator have already been proposed for gray box fuzzing to quantify uncertainty (see [22]). We hypothesize that the GTE also works in the context of IAST to quantify the uncertainty related to findings that cannot yet be identified as true or false positives.

Thanks to the targeted generation of test cases (as discussed in Sects. 6.5.2 and 6.5.3) and the information about the type of a vulnerability that allows to employ vulnerability-specific test oracles, true positives can be specifically triggered and observed during test execution. Hence, true positives from the static analysis can be verified very well as discussed in the context of RQ1 and RQ2 in the previous sections.

Approaches such as directed fuzzing can also identify true positives quite well. However, the benefit of our IAST approach over directed fuzzing is that we can assure at least for some findings from the static analyses that they are indeed false positives. Since we employ constraint solving to assess if a path that would execute a vulnerability is feasible, i.e., its path conditions are satisfiable, we can conclude that the potential vulnerability identified by the static analysis is a false positive if no path related to a specific vulnerability is feasible.

Even though it may not be possible to clearly group all findings into true and false positives, we can achieve a certain preliminary filtering which allows to focus on the remaining findings. For those, we would like to estimate how certain it is that they are false positives even in the case a certain number of test cases did not expose them. Our hypothesis is that the GTE applied on the context of gray box fuzzing is helpful also in this context and allows us to draw conclusions on the status if a reported vulnerability is a false positive.

The GTE values recorded during the experiments are intended to provide information on how well the GTE is suitable to draw such conclusions. If we could do this, the GTE can serve as a stop condition that would provide us the information when it is unlikely we will identify the vulnerability in question and, thus, allows us to stop the dynamic analysis.

Figure 6.5 shows the plots of the GTE values during the general fuzzing campaign performed by AFL/AFLGo. Both plots show that after the first bunch of test cases, the value drops to near or equal to zero.[18] The following peaks indicate that new execution paths have been found. The GTE progression for AFLGo clearly shows that after 3,000 test cases the GTE value gradually decreases and is close to zero for a short point in time before it increases again. Shortly before the end

[18] Zero means there is currently no execution path that has been observed only once.

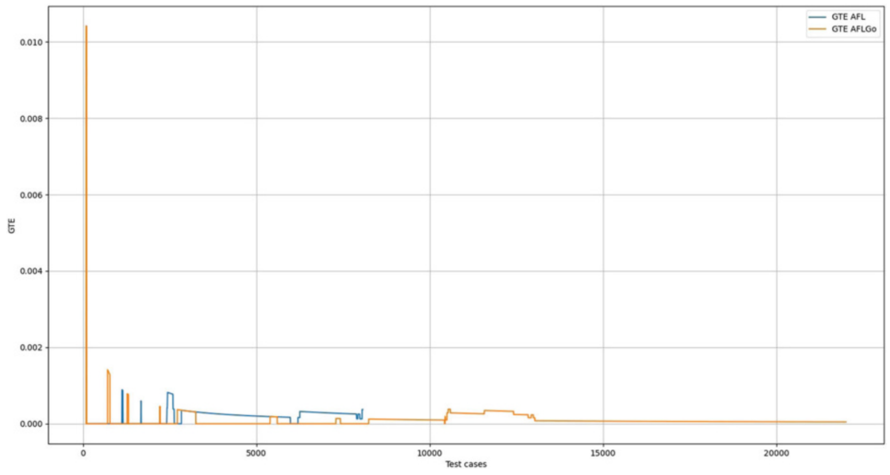


**Fig. 6.5** GTE plot for gray box/directed fuzzing Heartbleed Bug

of the recording, several jumps can be observed that indicate that new paths were executed that finally lead to the Heartbleed Bug. However, for the directed fuzzing using AFLGo, the value decreases very slowly after 13,000 test cases until the end of the recording without major changes. This long strictly monotonously decreasing course of the GTE value is here the advantage we seek for. Due to its stability in contrast to general fuzzing, the assumption would be gained that we achieved a stop criterion since the course of the values does not have any further spikes. However, this raises the question of whether the GTE is at all suitable as a stop criterion. The experiments performed seem to question this; however, this needs to be more thoroughly investigated in further experiments.

In summary, we can conclude that we can well identify true positives from the static analysis. Moreover, we can identify some of the false positives with certainty what is an advantage over gray box fuzzing and directed fuzzing. However, even in the case where we apply directed fuzzing to analyze a potential vulnerability and we are not able to trigger it, the GTE provides us with a means to quantify the uncertainty related to the discovery probability, and thus, it may serve as a stop condition.

### *6.5.5 Threats to Validity*

To account for the randomness of fuzzing, we have repeated every experiment 30 times and then analyzed the average results. Furthermore, we have chosen a real-world vulnerability in a heavily used SUT to analyze the potential of our approach. This should provide a more realistic demonstration compared to one which is based on automatically constructed bugs such as LAVA-M [37]. The metrics that we have analyzed were specifically chosen to compare the efficiency of the bug discovery, i.e., number of required test cases and time to exposure. Broader metrics, e.g., overall coverage, would not provide much insight into this question. To investigate the differences in branch discovery over time, we have analyzed the GTE. This might be a more difficult metric, as it is only a statistical estimator. However, it still can be useful as it provides a scientific approach to estimate probabilities that otherwise could be difficult to obtain.

Nevertheless, there are some downsides to our approach. First, our evaluation is limited to only one SUT and one vulnerability. This reduces the degree to which one can generalize the results to other contexts. Furthermore, our tools are currently in a prototypical stage which requires some manual updates to work on OpenSSL. Our implementation might thus be particularly fitted for the requirements of the Heartbleed Bug. Finally, we have also not performed a statistical test to estimate whether our observations would generalize to the overall population.

## 6.6 Conclusion, Limitations, and Outlook

Combining static and dynamic analysis within IAST can improve both analysis methods by exchanging information which cannot be determined by the respective analyses themselves. It could be shown that IAST approaches can be more efficient in finding vulnerabilities than using static and dynamic analysis separately. Moreover, we proposed to use a statistical means, i.e., the Good-Turing estimator, to enable the dynamic analysis of potential false positives from the static analysis. Together with constraint solving, we are able to confirm false positives partially, what dynamic analysis is not able to do on its own. For the remaining findings from the static analysis, the Good-Turing estimator serves as a means to decide when to stop the dynamic analysis of a potential vulnerability and consider it as a false positive. Even if a number of tests have to be performed in the case of suspected false positives, this should not be considered as having a negative impact on IAST performance. The targeted testing of the static analysis results and the reporting of the residual risk are clear advantages of the IAST approach over manual verification.

All experiments have been performed along the well-known Heartbleed Bug. The authors are aware that the evaluation along a single vulnerability does not allow to generalize the conclusions. Furthermore, it could not be analyzed how well the approach scales. This applies in particular to the verification of static analysis findings. If the number of false positives is high, IAST may lose its advantage since it would spend much effort on the identification of false positives than on discovering actual vulnerabilities.

Our future work will focus on the analysis of the efficiency of IAST compared to DAST to identify more accurately the conditions when gray box fuzzing approaches are sufficient and when IAST is of advantage. Furthermore, we plan to identify more sophisticated approaches on the application of statistical means related to false positives. At the time of writing, our prototypical implementation can handle magic bytes for integers and generate values for them. Our next plans are to extend the magic bytes to other data types (e.g., structs) and to experiment with supporting fuzzing campaigns by directing them with magic bytes. We will spend further effort on the exploitation of additional information from the static analysis to improve the dynamic analysis beyond the verification of findings from the static analysis and vice versa, i.e., providing further runtime information to the static analysis. Furthermore, our aim is to switch from the parallel IAST approach to the parallel iterative IAST approach to analyze how the iterative analysis improves both SAST and DAST mutually and incrementally. In this context, an open question might be when to stop iterative IAST and what is the overall benefit over the non-iterative IAST approaches. We will additionally investigate the effectiveness and efficiency of our approach on further systems and vulnerabilities.

**Acknowledgments** This work was supported by the Fraunhofer Internal Programs under Grant No. PREPARE 840 231.

## References

1. TIOBE, TIOBE Index (2022). https://www.tiobe.com/tiobe-index/. [Online; Accessed 03 Aug 2022]
2. N. I. of Standards and T. (NIST), CWE Over Time (2022). https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cwe-over-time. [Online; Accessed 03 Aug 2022]
3. B.P. Miller, L. Fredriksen, B. So, An empirical study of the reliability of unix utilities. Commun. ACM **33**(12), 32–44 (1990)
4. A. Takanen, J.D. Demott, C. Miller, A. Kettunen, *Fuzzing for Software Security Testing and Quality Assurance* (Artech House, 2018)
5. M. Schneider, J. Großmann, N. Tcholtchev, I. Schieferdecker, A. Pietschker, Behavioral fuzzing operators for UML sequence diagrams, in *International Workshop on System Analysis and Modeling* (Springer, 2012), pp. 88–104
6. M. Schneider, J. Großmann, I. Schieferdecker, A. Pietschker, Online model-based behavioral fuzzing, in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops* (IEEE, 2013), pp. 469–475
7. M. Zalewski, American fuzzy lop (2019). http://lcamtuf.coredump.cx/afl
8. L.D. Group, Libfuzzer – a library for coverage-guided fuzz testing (2019). https://llvm.org/docs/LibFuzzer.html
9. S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, H. Bos, Vuzzer: application-aware evolutionary fuzzing, in *NDSS*, vol. 17 (2017), pp. 1–14
10. Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, A. Tiu, Steelix: program-state based binary fuzzing, in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (ACM, 2017), pp. 627–637
11. N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, G. Vigna, Driller: augmenting fuzzing through selective symbolic execution, in *NDSS*, vol. 16 (2016), pp. 1–16
12. A.B. Chowdhury, R.K. Medicherla, R. Venkatesh, Verifuzz: program aware fuzzing, in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Springer, 2019), pp. 244–249

13. S.K. Cha, M. Woo, D. Brumley, Program-adaptive mutational fuzzing, in *2015 IEEE Symposium on Security and Privacy* (IEEE, 2015), pp. 725–741
14. J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, G. Vigna, Difuze: interface aware fuzzing for kernel drivers, in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (ACM, 2017), pp. 2123–2138
15. V.-T. Pham, M. Böhme, A. Roychoudhury, Model-based whitebox fuzzing for program binaries, in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)* (IEEE, 2016), pp. 543–553
16. M. Böhme, V. Pham, M. Nguyen, A. Roychoudhury, Directed greybox fuzzing, in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30–November 03, 2017*, ed. by B.M. Thuraisingham, D. Evans, T. Malkin, D. Xu (ACM, 2017), pp. 2329–2344
17. V.-T. Pham, M. Böhme, A.E. Santosa, A.R. Căciulescu, A. Roychoudhury, Smart greybox fuzzing, arXiv preprint arXiv:1811.09447 (2018)
18. K.M. Alshmrany, M. Aldughaim, A. Bhayat, L.C. Cordeiro, Fusebmc v4: Smart seed generation for hybrid fuzzing – (competition contribution), in *Fundamental Approaches to Software Engineering – 25th International Conference, FASE 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, (ETAPS)* 2022, Munich, Germany, 2–7 Apr 2022, Proceedings, ed. by E.B. Johnsen, M. Wimmer. Lecture Notes in Computer Science, vol. 13241 (Springer, 2022), pp. 336–340
19. L. Borzacchiello, E. Coppa, C. Demetrescu, FUZZOLIC: mixing fuzzing and concolic execution. Comput. Secur. **108**, 102368 (2021)
20. S. Ognawala, F. Kilger, A. Pretschner, Compositional fuzzing aided by targeted symbolic execution. CoRR, abs/1903.02981 (2019)
21. I.J. Good, The population frequencies of species and the estimation of population parameters. Biometrika **40**(3–4), 237–264 (1953)
22. M. Böhme, *STADS: software testing as species discovery*, vol. 27 (2018), pp. 7:1–7:52
23. M. Böhme, D. Liyanage, V. Wüstholz, Estimating residual risk in greybox fuzzing, in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Athens, Greece, 23–28 Aug 2021, ed. by D. Spinellis, G. Gousios, M. Chechik, M.D. Penta (ACM, 2021), pp. 230–241
24. M. Bozga, J. Fernandez, L. Ghirvu, Using static analysis to improve automatic test generation, in *Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000*, Berlin, Germany, March 25–April 2, 2000, Proceedings, ed. by S. Graf, M.I. Schwartzbach. Lecture Notes in Computer Science, vol. 1785 (Springer, 2000), pp. 235–250
25. O. Chebaro, N. Kosmatov, A. Giorgetti, J. Julliand, Program slicing enhances a verification technique combining static and dynamic analysis, in *Proceedings of the ACM Symposium on Applied Computing, SAC 2012*, Riva, Trento, Italy, 26–30 March 2012, ed. by S. Ossowski, P. Lecca (ACM, 2012), pp. 1284–1291
26. X. Wang, H. Chen, Z. Jia, N. Zeldovich, M.F. Kaashoek, Improving integer security for systems with KINT, in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012*, Hollywood, CA, USA, 8–10 Oct 2012, ed. by C. Thekkath, A. Vahdat (USENIX Association, 2012), pp. 163–177
27. H. Liang, L. Wang, D. Wu, J. Xu, MLSA: a static bugs analysis tool based on LLVM IR, in *17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD 2016*, Shanghai, China, May 30–June 1 2016, ed. by Y. Chen (IEEE Computer Society, 2016), pp. 407–412
28. N. Williams, B. Marre, P. Mouy, M. Roger, Pathcrawler: automatic generation of path tests by combining static and dynamic analysis, in *Dependable Computing – EDCC-5, 5th European Dependable Computing Conference*, Budapest, Hungary, 20–22 Apr 2005, Proceedings, ed. by M.D. Cin, M. Kaâniche, A. Pataricza. Lecture Notes in Computer Science, vol. 3463 (Springer, 2005), pp. 281–292

29. P. Godefroid, M.Y. Levin, D.A. Molnar, Automated whitebox fuzz testing, in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008*, San Diego, California, USA, 10–13 Feb 2008 (The Internet Society, 2008)
30. P. Godefroid, M.Y. Levin, D.A. Molnar, SAGE: whitebox fuzzing for security testing. ACM Queue **10**(1), 20 (2012)
31. T. Ball, The concept of dynamic analysis, in *Software Engineering – ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Toulouse, France, Sept 1999, Proceedings, ed. by O. Nierstrasz, M. Lemoine. Lecture Notes in Computer Science, vol. 1687 (Springer, 1999), pp. 216–234
32. E.W. Dijkstra et al., *Notes on Structured Programming* (1970)
33. P.D. Schubert, B. Hermann, E. Bodden, Phasar: an inter-procedural static analysis framework for c/c++, in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Springer, 2019), pp. 393–410
34. L.D. Moura, N. Bjørner, Z3: An efficient SMT solver, in *International conference on Tools and Algorithms for the Construction and Analysis of Systems* (Springer, 2008), pp. 337–340
35. M. Böhme, V.-T. Pham, M.-D. Nguyen, A. Roychoudhury, Directed greybox fuzzing, in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (ACM, 2017), pp. 2329–2344
36. A. Arcuri, L. Briand, A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Softw. Test. Verif. Reliab. **24**(3), 219–250 (2014)
37. B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W.K. Robertson, F. Ulrich, R. Whelan, LAVA: large-scale automated vulnerability addition, in *IEEE Symposium on Security and Privacy, SP 2016*, San Jose, CA, USA, 22–26 May 2016 (IEEE Computer Society, 2016), pp. 110–121

# Part III
# Protection at Operations

# Chapter 7
# CTAM: A Tool for Continuous Threat Analysis and Management

**Laurens Sion, Dimitri Van Landuyt, Koen Yskout, Stef Verreydt, and Wouter Joosen**

**Abstract** Security and privacy threat modeling approaches are commonly applied to identify and address design-level security and privacy concerns in the early stages of software development. Identifying and mitigating these threats should remain a continuous concern during the development lifecycle, as single-shot analyses become quickly outdated with contemporary agile development practices. Despite expert recommendations, the support for continuously applying these types of approaches throughout the development lifecycle is limited. In this article, we present an integrated threat analysis toolchain for automated, continuous threat elicitation, assessment, and mitigation as part of continuous integration pipelines in the GitLab DevOps platform. Automating the threat analysis enables continuous attention to security and privacy threats during the development and supports monitoring and managing the progress in mitigating security and privacy threats over time. Additionally, the integration of threat analysis in a continuous integration pipeline enables more advanced and fine-grained analyses such as assessing the impact of proposed changes in feature branches and the analysis of merge/pull requests for their impact on the threat model. We evaluate the approach and its prototype on a concrete real-world application to assess the threat analysis of multiple application versions over time as changes are made and new features introduced. We conclude with an in-depth discussion on the use of threat modeling in continuous integration contexts.

This is an extended version of an earlier published conference paper: Laurens Sion, Dimitri Van Landuyt, Koen Yskout, Stef Verreydt, Wouter Joosen, Automated Threat Analysis and Management in a Continuous Integration Pipeline, 2021 IEEE Secure Development Conference (SecDev 2021), Online, October 18–20, 2021 [46].

L. Sion (✉) · D. Van Landuyt · K. Yskout · S. Verreydt · W. Joosen
imec-DistriNet, KU Leuven, Leuven, Belgium
e-mail: Laurens.Sion@cs.kuleuven.be; Dimitri.VanLanduyt@cs.kuleuven.be; Koen.Yskout@cs.kuleuven.be; Stef.Verreydt@cs.kuleuven.be; Wouter.Joosen@cs.kuleuven.be

A. Sadovykh et al. (eds.), *CyberSecurity in a DevOps Environment*,
https://doi.org/10.1007/978-3-031-42212-6_7

**Keywords** Threat modeling · Threat analysis · DFD · Security by design · Privacy by design · Threat management

## 7.1 Introduction

Security and privacy require continuous attention throughout the software development lifecycle (SDLC). It is well-known, though, that absolute security cannot be achieved and compromises must be made. In practice, security efforts should therefore be directed toward conscious management of risk and security debt [36], a form of technical debt. Without sufficient attention to security, the security debt and risk may increase beyond acceptable levels, increasing the likelihood of security incidents and associated losses and making it hard to recover without major investments and delays.

Security and privacy threat modeling techniques [13, 39, 40, 58] are typically applied in the early phases (requirements and design) of the SDLC. The importance of addressing design security is further emphasized by the recent inclusion of *insecure design* in the 2021 OWASP top 10 [31]. These threat modeling approaches reason at an abstract level about the system, often in the form of a data flow diagram (DFD), to elicit many potential security threats. In a next step, mitigations for the most important of these threats (in terms of risk) are selected, which can then be incorporated during the software development.

Current threat modeling approaches are not well-aligned with contemporary development practices. Modern software development happens at a fast pace with frequent changes to the code base to introduce new functionality, fix bugs, and refactor the design. Continuous integration (CI) is one of the enablers of this fast pace. Threat modeling, on the other hand, is often a manual, time-consuming, one-off (or infrequently repeated) activity conducted in workshops involving experts and numerous stakeholders [54, 60]. Reliance on extensive manual labor prohibits frequent re-evaluation as the software design evolves. This in turn is considered problematic, because the goal of threat modeling is precisely to identify threats that carry a significant risk, and (because they are linked to the design) that may be hard to mitigate afterward.

An additional problem associated with infrequently revisiting a threat model is that it hampers adequate management of the risk and security debt as part of project management. Indeed, effective decision-making relies on having a clear view on the current status and progress, the impact of the possible choices, and the effectiveness of past decisions and efforts. Infrequent threat modeling only yields a coarse-grained view on the progress that is being made, though, precluding swift reactions to emerging risks.

Recently, a number of threat modeling tools and approaches have emerged that provide a degree of automation and increase the repeatability of the threat assessment [50] even as the system evolves. These tools however are often still standalone, in the sense that they lack integration with widely used development and project management platforms typically used in the context of continuous integration.

This chapter introduces CTAM (continuous threat analysis and management), a novel approach and corresponding toolchain that addresses these problems by technically integrating an automated threat analysis and assessment activity in a continuous integration pipeline. This enables stakeholders to monitor threat modeling results and track and manage the evolution of risk based on information that evolves together with the implementation. CTAM leverages the possibilities offered by automated threat modeling tools to achieve traceable, systematic, and frequent reassessments. The input for an existing threat modeling tool, which consists mainly of a model-based representation of the system (e.g., a DFD), is placed and maintained alongside the source code in a version control system. The automated threat modeling tool is then used as a standalone analysis engine in a continuous integration job. By combining the automated analysis results with the existing version information from the repository, the current state and historic evolution of the threat model, as well as the impact of suggested modifications in different branches, can be assessed and presented on a dashboard. This information can subsequently be used by developers and security experts to aid decision-making, the instantiation of appropriate countermeasures, and tracking the effectiveness of these countermeasures over time.

This chapter introduces the following contributions: (i) it presents the CTAM approach and toolchain, leveraging existing threat modeling automation enablers in support of systematic, automated, and continuous threat analysis and management; (ii) it presents a prototype implementation and validates the prototype on a research and a real-world application case (the backend system of a contact tracing application), demonstrating its capability to recognize a number of different risk evolution patterns; and (iii) it provides an in-depth discussion on automated threat modeling as part of a continuous integration pipeline.

In addition to being a practical tool to continuously monitor and manage threat-centric risk during development, the CTAM toolchain is a technological enabler for continued research toward more advanced analysis techniques, for example, to retroactively study the emergence and management of risk in real-world code bases.

This chapter is an extension of earlier work [46]. More specifically, the following extensions are provided: (i) extending the description of CTAM and the inputs for the analysis, (ii) adding an in-depth evaluation on 12 versions of a real-world application, (iii) providing a more extensive discussion on version granularity, model granularity, and model scope, and (iv) adding a roadmap on (automating) model reconstruction, runtime monitoring, and longitudinal project management.

The chapter is structured as follows: Section 7.2 describes the related work on threat modeling and continuous quality assessment. Next, Sect. 7.3 presents CTAM and the implementation aimed at demonstrating its feasibility. Then, Sect. 7.4 applies CTAM on an application case of software-as-a-service (SaaS) document generation and delivery platform and illustrates the type of analysis that it enables. Section 7.5 applies CTAM on 12 versions of a real-world application that provides backend for a contact tracing application. Afterward, Sect. 7.6 provides a discussion on the use of CTAM, including the possibility of using other threat elicitation engines, the importance of the consistency of the model with the source code, and additional analysis types that become available with CTAM. Subsequently, Sect. 7.7 provides a roadmap on future work on model reconstruction, runtime monitoring, and longitudinal project management. Finally, Sect. 7.8 concludes the chapter.

## 7.2 Related Work

Continuous integration refers to software development practices that are centered heavily around a central version control system and code repository. These systems implement a pipeline of automated activities that are typically aimed at quality control (e.g., code style checking), automated testing (e.g., regression testing, integration testing, acceptance testing), and automated building and build management. Automating these activities allows for frequent execution at the level of individual code commits, providing the developer with rapid feedback and shortening the time to address issues. These key principles enable ensuring a certain degree of quality assurance.

This section outlines the related work on threat modeling in this context. First, the current state of the threat modeling support during development is discussed. Next, the state of the art in continuous integration and security analysis activities in this context is outlined.

### *7.2.1 Threat Modeling Support During Development*

Several threat modeling tools and approaches, such as IriusRisk [19] and Autodesk CTM [1], promote the integration of threat modeling during development specifically by linking threats to issues in an issue tracker. While this enables tracking the progress regarding the identified security and privacy threats, the threat mitigation progress is monitored in the issue tracker, rather than in the system model. Furthermore, while such approaches may support versioning of the system model, they do not support analysis of a threat model over time.

More closely aligned with the source code is ThreatSpec [53]. It provides a set of code annotations that can assist in constructing and maintaining a DFD model by inserting comments at the relevant locations in the source code. ThreatSpec does not perform any threat elicitation by itself, so the extracted model will have to be analyzed manually or with another tool to obtain a list of threats to further analyze and aggregate. It does allow documenting threats and mitigations through code annotations so that the results of the threat elicitation activity can be captured as well.

Pytm [51] generates diagrams (DFDs and sequence diagrams) and threats based on a system model expressed in Python code. Such a representation enables versioning the system model together with the source code. It does not, however, provide risk estimates for threats, so monitoring progress in terms of risk reduction requires additional analysis.

SPARTA [49] is an eclipse-based threat modeling tool that automatically elicits security and privacy threats based on XMI files of solution-enriched DFDs and threat catalogs (e.g., the STRIDE [40] and LINDDUN [47] threat types are supported). SPARTA automates the risk analysis of individual threats and supports calculating the aggregate values for the system, which is required for monitoring the threat mitigation progress. SPARTA does not support any historical analysis of the threat mitigation progress. While SPARTA provides both a graphical DFD model editor and the engine for eliciting security and privacy threats, its elicitation engine can also be run standalone on the DFD model files. This motivates the adoption of the SPARTA engine in the context of CTAM.

Threagile [38] generates threat model reports based on YAML files of the architecture and its assets and provides pipeline integration to do so in a continuous fashion. However, such analyses are only focused on a single version of the system; it does not analyze how those threat models evolve over multiple versions of the system. A very similar and recently released tool is TicTaaC [37], which also relies on one (or more) YAML files describing the data flow model. TicTaaC generates a report from this with the findings. Analogous to Threagile, it is intended to run as part of a CI pipeline, but its analysis is also focused on single version of the system.

OWASP's Threat Dragon [32] is an open-source threat modeling platform for system modeling and threat elicitation. Its documentation mentions that future versions should provide an API for pipeline integration, but this is not supported at the time of writing.

### 7.2.2 *Quality Assessment in Continuous Integration Pipelines*

Several approaches exist that conduct frequent code analysis for measuring the impact on qualities such as performance, maintainability, security, etc. For example, the PerfCI tool [20] integrates automated performance benchmarks to identify potential performance regressions over time. Vassallo et al. [56] in turn presented an approach that automates and integrates the identification of bad practices, anti-patterns, or common misconfigurations in a CI pipeline.

The automation of these activities is a key enabler for extensive data analytics at the level of the code base: the evolution of a code base in terms of software quality can be monitored and evaluated [22] over longer periods of time.

Static code checkers (SAST) allow for the identification of vulnerabilities as a result of code-centric analysis [12, 41]. As discussed by Rangnau et al. [35], integrating dynamic security testing (DAST) is more challenging as these more advanced analysis techniques incur a more significant performance cost, to the extent that the total cost of their integration in a CI pipeline might become prohibitive.

To our knowledge, model-based analysis activities that identify threat scenarios at the level of an abstraction model of the system (i.e., threat modeling and threat-based risk assessment) have not yet been integrated in a practical CI pipeline, with the exception of Threagile [38] which does not consider the analysis over time. Yet, threat modeling experts and advocates strongly encourage frequent re-evaluation of the outcome of a threat modeling and analysis exercise [3, 40, 43, 55] throughout the development of a system. In this article, we present the practical implementation of such an activity in the GitLab DevOps platform [18].

## 7.3 Continuous Threat Analysis and Management

The main goal of CTAM is to automate continuous threat analysis, management, and progress monitoring by integrating it in continuous integration pipelines. This is achieved by (1) storing the model together with the source code in version control (this model contains the DFD, the applied security and privacy solutions, and inputs for the risk analysis); (2) for every push to the repository, running a continuous integration analysis job to elicit security or privacy threats and perform a risk analysis on them; (3) collecting and aggregating these results in the CTAM server; and (4) making these results available as feedback to the developers. Figure 7.1 provides a graphical overview of the approach. The next subsections will elaborate

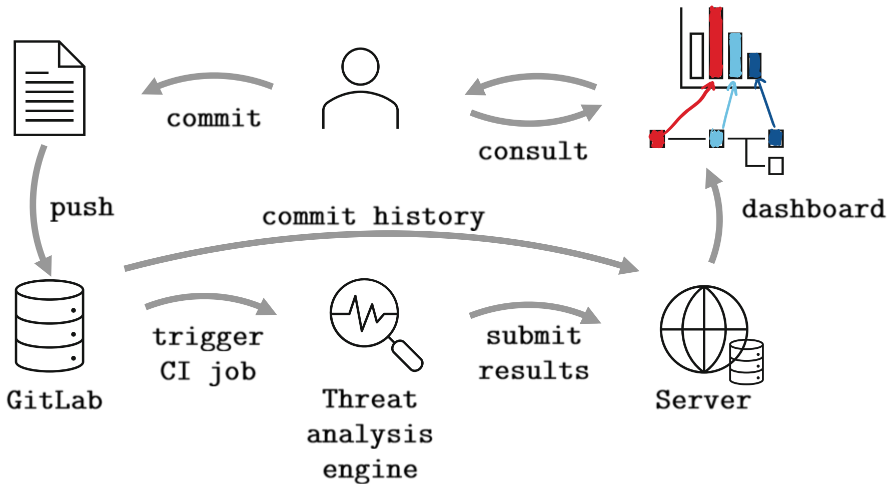


**Fig. 7.1** Overview of the approach. On the left-hand side, changes to the codebase and model are committed and pushed to a repository on GitLab. This triggers the CI jobs that will run the threat analysis engine (bottom center), of which the results will be submitted to the server. Finally, the developers can consult the impact of their changes on a dashboard presenting the analysis results (right-hand side)

on the necessary inputs (1), the threat analysis (2), the types of analysis activities offered by CTAM (3), and the implementation of the CTAM server (4).

## *7.3.1 Threat Analysis Inputs*

The required inputs vary based on the chosen analysis engine. For the individual threat analysis, CTAM currently leverages the SPARTA [45] threat modeling engine (Sect. 7.3.2). All the relevant data (i.e., the DFD model, solutions, attacker profiles, and threat-type catalogs) for SPARTA's analysis are contained in one (or more) model files that will be read by the analysis engine. The required inputs for the SPARTA engine are discussed shortly here.

### 7.3.1.1 DFD Model

The main input is the DFD model of the application under consideration. This model contains the processes, data stores, external entities, data flows between these elements, and trust boundaries. To support the risk-driven prioritization of the elicited security and privacy threats, elements of the DFD model can be extended with asset values to specify the loss magnitude or damage that would arise when threats would manifest themselves at these elements. The SPARTA engine will leverage these to calculate the impact of a threat [48].

### 7.3.1.2 Security and Privacy Solutions

As applicable, the model can be extended with security and privacy solutions that are applied to particular elements of the system to mitigate threats at these locations. These solutions specify the involved elements, the protected elements, and which threats they mitigate. For example, a logging solution could specify a data store which holds the logs and a process which logs incoming requests to that data store, so that repudiation threats to that process are mitigated. Furthermore, a solution also includes information on its strength, which the SPARTA engine will compare against the attacker profile (explained below) to calculate the likelihood of a threat.

### 7.3.1.3 Attacker Profiles

For the analyses, one (or more) attacker profiles can be used. These profiles express the capabilities of different types of adversaries, how frequently these adversaries come into contact with the system, and to which elements of the system (if any) they have insider access. These profiles are taken into account during the risk assessment of the elicited threats. For example, different attacker profiles could be defined for script kiddies and more advanced adversaries. The capabilities of these different types of adversaries are compared against the strength of the countermeasures to determine whether an attacker is able to defeat a certain countermeasure or if it is sufficiently strong against the considered types of attackers.

#### 7.3.1.4 Threat-Type Catalog

Finally, the particular threat-type catalog to use is specified in the model. Such a threat-type catalog contains the list of threat types that need to be elicited and the criteria to use to determine whether a threat of that particular type is applicable. These threat-type catalog resources can also be extended and customized as desired.

## *7.3.2 Threat Analysis Engine*

SPARTA enables (i) the automated generation of threats at the basis of a (customizable) threat catalog; (ii) per threat, a risk estimation step [48] that takes into account many factors documented in the input model (e.g., the application of security solutions [49], a description of the affected data subjects in case of privacy [44]); and, finally, (iii) the aggregation and disclosure of these outcomes.

Threats are elicited by performing model queries on the supplied model. The threat-type catalogs contained in the model specify the criteria for the threats to be applicable and can be used to encode, for example, element- or interaction-based STRIDE threats as well as more complex threat patterns. As the main input of the analysis is the model of the system under development, the scope of the analysis is necessarily limited to the design of this system. During the analysis, SPARTA generates relevant threats linked to the elements in the DFD model (e.g., spoofing external entity A in the data flow to Process B).

For the risk analysis and prioritization, SPARTA processes the information in the model (i.e., asset value, strength of security solutions, etc.) to determine how effective the countermeasures are to protect against the elicited threats. The resulting value is the expected loss (impact $\times$ likelihood), expressed in the same unit as the asset value. It is up to the developers or business stakeholders to provide this information in a unit that is convenient to them.

We created a dockerized version of the SPARTA threat elicitation and assessment engine, which reads a configuration file (specifying the model file and the submission server), analyzes the model (i.e., elicits security or privacy threats and performs a risk assessment of these threats), and submits the threat elicitation results to the submission server. The bottom of Fig. 7.1 depicts these steps graphically. The threat analysis engine in the center runs on the last commit, analyzes the model contained therein, and submits the results.

The docker container enables the use of SPARTA in GitLab CI jobs [18].[1] The only additional information required in the repository is the DFD model file and the aforementioned configuration file. Because individual commits are analyzed, that model file will need to accurately reflect any changes that are made to the codebase. Section 7.6 further discusses the need for an accurate model of the system under analysis.

The actual integration of the threat elicitation and analysis in the CI pipeline is straightforward, as it only requires a build step that runs a docker container in the checked out repository. The following code fragment illustrates the additional analysis build job that needs to be added to the CI pipeline configuration:

```
threat_analysis_job:
  image: sparta-docker:latest
  stage: build
  script:
    - sparta
```

Finally, it is possible to use other threat modeling tools for the elicitation, as long as they yield appropriately formatted threats and their corresponding risk estimates for submission to the CTAM server (see Sect. 7.6.1 for a discussion on the use of alternative threat elicitation engines).

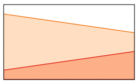
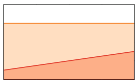
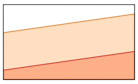
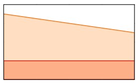
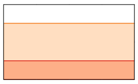
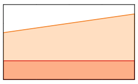
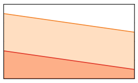
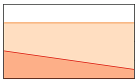
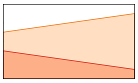
### 7.3.3 *Analysis Activities*

CTAM enables several types of analysis activities through its systematic collection of threat analysis information as the software system evolves over time. It currently leverages SPARTA for the threat analysis results and hence also relies on the residual and inherent risk values that SPARTA provides. Inherent risk represents the risk not accounting for any security or privacy solutions (i.e., the risk of a threat if there were no countermeasures at all). As such, this is the degree of risk inherent to the nature of the system under design. Residual risk represents the risk taking into account security and privacy solutions (i.e., the inherent risk minus the effect of security and privacy solutions). However, it is also possible to use different risk scores, as long as they incorporate the effect of partially or completely mitigating threats in the system.

By collecting the inherent and the residual risk for every committed version of the system under consideration, the overall progress in securing the system can be assessed. Table 7.1 presents an overview of the different risk evolution patterns that may emerge through the combination of a decrease, stable, or increase of the

[1] While we leverage GitLab CI, adapting the approach to run in the context of GitHub actions is straightforward.

**Table 7.1** Risk evolution patterns. Plots for the different risk evolution patterns due to decreasing, stable, or increasing inherent and residual risk values. The area plots are not stacked (i.e., the inherent risk consists of the entire area under the line including the residual risk). Combinations of these patterns are possible to express to different slopes of the inherent and residual risk plot lines. For example, combining VII and IV results in a more slowly decreasing residual risk, combining III and VI in a more slowly increasing residual risk, etc.

| Residual Risk | Inherent Risk *(top line)* | | |
|---|---|---|---|
| *(bottom line)* | Decrease | Stable | Increase |
| Increase | I  Remove security solutions[*] | II  Remove security solutions | III  Add insecure functionality |
| Stable | IV  Remove secure functionality | V  No security-relevant changes | VI  Add secure functionality |
| Decrease | VII  Remove insecure functionality | VIII  Add security solutions | IX  Add security solutions[*] |

[*] Solutions that introduce additional risk with regard to, for example, cryptographic key material

inherent and residual risk values.[2] These patterns allow developers to gain insight into the progress that is being made over time. For example, these patterns will show which commits focus on security (reducing the residual risk) or on expanding functionality without considering security (increased residual risk). It will also show how these types of changes manifest themselves over time (e.g., whether security is always considered after new functionality has been introduced or at the same time).

Alternatively, rather than merely tracking the overall security progress, CTAM can also be leveraged proactively in multiple ways: First, the security impact of one or more changes (i.e., pull or merge requests) can be analyzed and compared to the main branch before they are merged. Second, multiple variants of a proposed change can be analyzed and compared when deciding which one would be best in terms of security. Third, the analysis results can be used to automatically reject pull requests if the increase in risk surpasses a certain threshold. This is not yet explicitly

[2] While Table 7.1 shows those patterns as nine distinct possibilities, there is actually a continuum as one sort of risk may, for instance, decrease more rapidly than the other one. For example, if both secure and insecure functionality (cells VII and IV in Table 7.1) are removed from the system in a single commit, both the residual and the inherent risk plot lines will decrease, but the residual risk line will have a shallower slope than the inherent risk line.

visualized in the prototype, but all the necessary analysis information is available to CTAM to show this.

### *7.3.4 Server*

The server component is a Spring boot application. Registering a new project requires a deployment token and the repository URL. This is used by the server to retrieve the commit history from GitLab. When analysis results are submitted by the threat analysis engine, these results are associated with the corresponding git commits to enable the construction of an overview dashboard (depicted in the right-hand side of Fig. 7.1).

When a developer consults the CTAM dashboard, the server constructs a historical overview of the evolution of the aggregated risk by combining the analysis results for the commit ancestors on the main branch of the repository. This comprises the following calculations per commit: (i) the threat count, (ii) the total inherent risk (by aggregating the inherent risk of the individual threats), (iii) the total residual risk (also by aggregating), (iv) the risk reduction (as the mitigated risk over the inherent risk), and (v) the classification into categories (by binning the threats in equal intervals based on the largest inherent risk encountered in the analysis of the commit). This initial set of measurements can be expanded with additional ones that can be calculated from the submitted threat results such as the most frequently occurring threat types, the system elements with the largest residual risk, etc.

Figure 7.2 shows the CTAM dashboard containing information on the evolution of threats, the estimated risk, and the progress in reducing that risk for a specific project. In addition to a project-wide overview, the developer can also select any analyzed commit from the overview to obtain the detailed analysis results for that specific version of the system, including the full list of elicited threats.

## 7.4 Functional Validation

This section presents the functional validation of CTAM on an illustrative application case to demonstrate how the effect of changes can be perceived in the aggregated risk analysis. First, the application itself is described. Next, a number of deliberate change scenarios are introduced to assess the effect of different types of changes (e.g., new functionality, securing existing functionality). After each of these changes, the resulting model is analyzed, and the analysis results are collected. Finally, the results for each change scenario are discussed, highlighting the usefulness of CTAM in measuring and monitoring the security impact during software development.

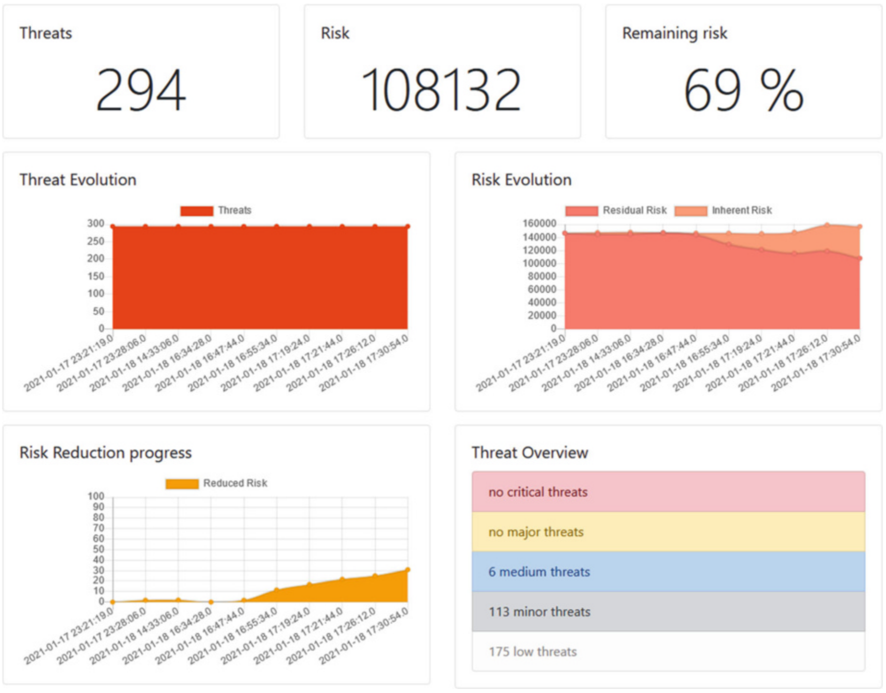
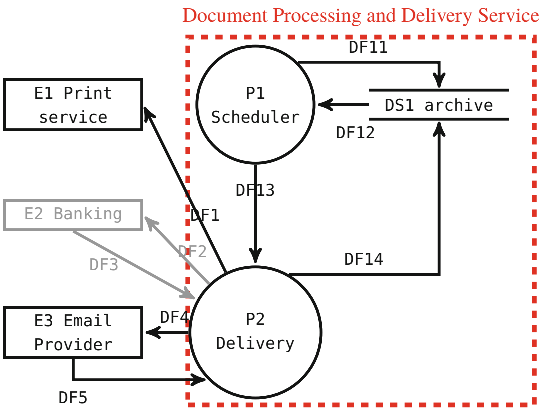


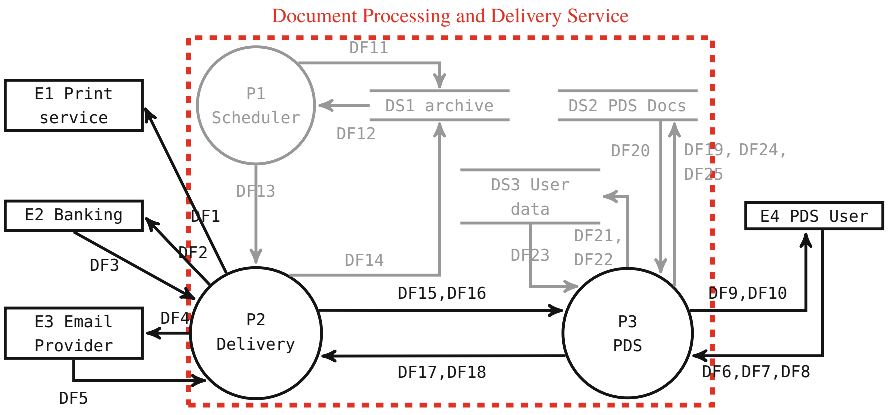
**Fig. 7.2** CTAM dashboard. The dashboard presents the main metrics of the last commit (top row), the evolution of the number of threats and the residual and inherent risk (middle row), and the progress in reducing the risk and the overview of the prioritized threats in the last analysis results (bottom row)

### *7.4.1 Description of the Case*

We apply our prototype on a SaaS application for generating and delivering PDF documents (e.g., invoices, pay slips), via different delivery channels (e.g., email, print) to end users. One of those channels is a hosted personal document store (PDS) on which users can login to retrieve documents sent to them. Figure 7.3 shows the DFD of this system. The center part of the figure contains the core of the system's delivery services. The left-hand side contains the integration with third parties for delivery via print, email, etc. The right-hand side models the hosted PDS from which users can directly access their documents. The next section will refer to this diagram when explaining the different changes that will be made to this system to validate the CTAM prototype.



(a) DFD for versions C0–C2



(b) DFD for versions C3–C5

**Fig. 7.3** Data flow diagram (DFD) of the document processing and delivery service. This diagram shows the delivery components of this system together with the storage in the center of the diagram. The left-hand side shows various third-party delivery services, while the right-hand side shows the hosted personal document store (PDS) from which users can retrieve the documents sent to them. To improve the readability, multiple flows in the same direction are combined together (e.g., *DF17*, *DF18*). Figure 7.3a shows the initial version of C0 and the change of C1 in gray (C2's security solutions are not visualized). Figure 7.3b shows the added functionality of C3, and the elements that are removed in C5 in gray (C4's security solutions are not visualized)

## 7.4.2 *Change Scenarios*

We validate our approach with five specific change scenarios (affecting both functionality and security solutions). Each of these changes is applied to the DFD model of the document processing system in separate commits to enable the analysis of their impact. The security solutions mentioned below include encryption, authentication, and access control to protect against information disclosure, tampering, and spoofing:

C0 The initial version of the system does not contain the PDS functionality (i.e., no E4, DS2, DS3, P3, nor any of the data flows to or from them), nor the banking integration (i.e., no E2 or any of its data flows), nor any security solutions to protect the communication with E3.
C1 The first commit introduces secure functionality by adding the banking integration (E2) together with some security solutions to protect the communication with E2. These solutions provide encryption (to prevent information disclosure and tampering of the data flows) and mutual authentication.
C2 This commit exclusively affects security, by introducing a security solution to protect the communication with the email provider (E3). The solution provides encryption (to prevent information disclosure and tampering of the data flows) and authentication of the email provider.
C3 This commit adds the PDS functionality (i.e., E4, DS2, DS3, P3, and the data flows) but does not introduce any security solutions to secure this functionality.
C4 This commit adds security solutions to protect the communication between the PDS users (E4) and the PDS by encrypting the traffic (protection against information disclosure and tampering), authenticating the PDS for user registration, and mutual authentication for retrieving documents. This does not secure all the functionality introduced by C3.
C5 Finally, this commit removes all data stores and the scheduler (i.e., remove P1, DS1–3, and their data flows).[3]

Each of these changes is introduced in separate commits and pushed to a GitLab instance to trigger the continuous integration jobs which analyze the modified DFD model and submit the analysis results to the CTAM server.

### 7.4.3 Results

Figure 7.4 shows the analysis results as reported by the CTAM server after receiving the results from SPARTA for each of the introduced changes. This section revisits each of these changes to explain the risk evolution pattern encountered in the results and refers to the corresponding cells in Table 7.1.

C1 As shown in Fig. 7.4, the residual risk line is not perfectly stable: the change actually did result in an increase of the residual risk due to the fact that the solutions do not fully mitigate the total risk introduced by the new functionality. Hence, the change corresponds with cells VI and III in Table 7.1.

[3] While this is an unrealistic modification, it demonstrates the impact of removing insecure functionality from the system.

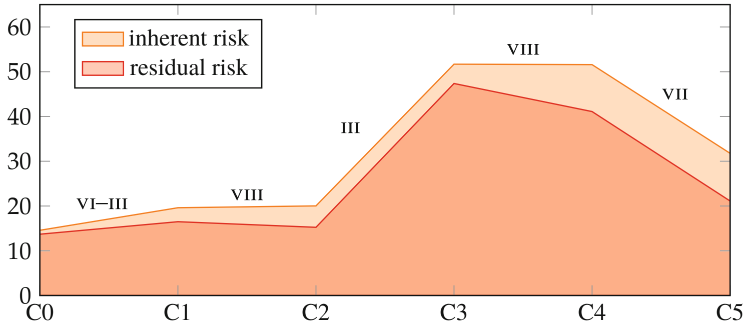


**Fig. 7.4** Overview of the analysis results for the different changes. This plot shows the resulting inherent and residual risk values in the example application for each of the changes

C2 With the exception of some small variance in the risk estimation, the inherent risk remains stable, while the residual risk is reduced. This corresponds with cell VIII in Table 7.1.

C3 This change scenario involves a substantial modification, as also visible from the analysis results. As this change scenario does not consider security, it results in both an increase of the inherent risk and the residual risk. As such, this is an example of the pattern in cell III in Table 7.1.

C4 As this change scenario only secures the interaction between the end user and the PDS, it does not mitigate all the newly introduced risk from the previous change scenario. As it only introduces security solutions, it again corresponds with cell VIII in Table 7.1.

C5 The final change removes insecure functionality from the model (all internal storage and the scheduling process). This results in a substantial reduction of both the inherent and the residual risk. This corresponds with cell VII in Table 7.1.

## 7.5 Evaluation

In addition to the functional validation, we also evaluate our approach on a concrete version history of an existing real-world implementation of a contact tracing application.

The Corona-Warn-App has been developed for Germany as privacy-friendly contact tracing app as part of the government's response against COVID-19. The app and its backend services are all open source [7–11], extensively documented (in English) [6], and use the Spring framework. This makes them a suitable candidate for creating models for different historical versions of the application and its corresponding backend services.

Following below is a brief description of the repositories that are taken into account for creating the models:[4]

cwa-server This repository [7] contains the server for processing upload requests from clients, the distribution configuration parameters to the mobile applications, and the aggregation and distribution (via a content delivery network (CDN)) of diagnosis keys.

cwa-testresult-server This repository [8] contains the test result server which is responsible for making available test results to the verification server.

cwa-verification-iam This repository [9] contains the identity and access management service to enable interaction with the verification server.

cwa-verification-portal This repository [10] contains the verification portal service to enable hotline employees to create proof certificates for users with a positive test.

cwa-verification-server This repository [11] contains the verification service that provides proof of positive tests to other components in the system.

## 7.5.1 *Modeling Approach*

We manually constructed a new DFD model of the entire application whenever one of these repositories introduces a new major or minor version. Figure 7.5 shows an overview of the different combinations of versions of the repositories that are used to construct 12 DFD models, representing the application as it is developed and extended.

To create the concrete DFD models, we based ourselves on a heuristic mapping from Spring annotations to DFD model element types:

@Repository A repository in the source code is translated to a data store in the DFD.

@Controller A controller is translated to a process.

@Service A service is translated to a process.

The data flows between these elements are added by inspecting the source code to determine whether there is communication between these elements. As the source code does not contain representations of end users or third-party integrations, these are manually added to the DFD based on the supporting documentation.

Finally, to support the risk analysis and illustrate the effect of countermeasures on the residual risk over time, a number of security solutions (e.g., access control, encrypted channels) are modeled as well. For the risk estimation process, we also need to assign value estimates to elements of the DFD. Because we do not have

[4] Understanding the inner workings of how these different components interact is not necessary for the evaluation in the next section. We refer the interested reader to the separate documentation repository [6] of the Corona-Warn project for more details.

12: 2020-10-28 s **v1.6.0**, ts **v1.1.1,** vi v1.1.0, vp v1.3.1, vs v1.3.2

11: 2020-09-22 s **v1.4.0**, ts v1.1.0, vi v1.1.0, vp **v1.3.1**, vs **v1.3.2**

10: 2020-08-19 s **v1.3.0**, ts v1.1.0, vi v1.1.0, vp **v1.3.0**, vs **v1.3.1**

9: 2020-07-16 s **v1.1.0**, ts **v1.1.0,** vi **v1.1.0**, vp **v1.1.0**, vs **v1.1.0**

8: 2020-06-12 s v1.0.1, ts **v1.0.0**, vi **v1.0.0**, **vp v1.0.0**, vs **v1.0.0**

7: 2020-06-08 s **v1.0.1**, ts **v0.6.0**, vi **v0.6.0**, vp **v0.6.0**, vs **v0.6.0**

6: 2020-06-05 s **v0.5.10**, ts **v0.5.0**, vi **v0.5.0**, vp **v0.3.2**, vs **v0.5.3**

5: 2020-05-31 s **v0.5.2**, ts **v0.3.2**, vi v0.3-alpha, vp **v0.3.1-alpha**, vs **v0.5.2**

4: 2020-05-28 s **v0.5.1**, ts **v0.3.1**, vi v0.3-alpha, vp v0.3-alpha, vs v0.3.1-alpha

3: 2020-05-27 s **v0.5.0**, vi **v0.3-alpha**, vp **v0.3-alpha**, vs v0.3.1-alpha

2: 2020-05-22 s **v0.4.0**, vs **v0.3.1-alpha**

1: 2020-05-14 s v0.3

**Fig. 7.5** Overview of Corona-Warn versions. This figure shows the overview of the relevant versions of the Corona-Warn repositories that are included in the model. Legend: s, cwa-server; ts, cwa-testresult-server; vi, cwa-verification-iam; vp, cwa-verification-portal; vs, cwa-verification-server. All versions that have changed are marked in **bold**

the domain knowledge of the actual application's stakeholders to provide realistic estimates for all elements, we make an approximation based on the solutions that were identified. Concretely, we assume that these solutions were introduced to protect the most important elements in the system and hence assign those elements a higher asset value than the other elements. This enables us to demonstrate the use of CTAM to assess risk evolution, yet with the caveat that our analysis does not provide an accurate estimate of the real-world risk associated with the application.

## 7.5.2 *Results*

The discussion on the results is split into two parts: The first part discusses the evolution of the model itself over time. The second part discusses the threat analysis results.

### 7.5.2.1 Evolution of the Model

The analyzed application demonstrates a fast-paced evolution over a time of about 5 months, as new functionality is introduced and it is further integrated with other services for processing test results and authentication. This increase in size and complexity is visible when plotting the counts of the different DFD model

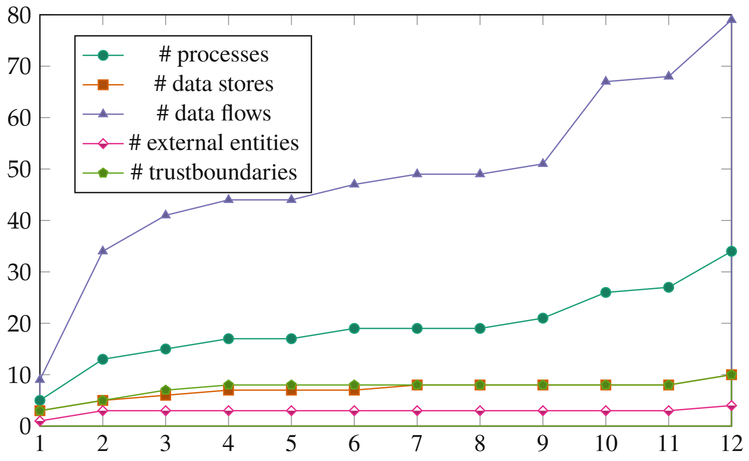


**Fig. 7.6** Overview of the model size evolution. This plot shows how the counts of different model element types change with each new version as the application increases in size and complexity

element types over time. Figure 7.6 shows the increasing DFD model size as more functionalities are included in later versions of the modeled application.

The graph also shows that the model itself does not converge during that time (especially in terms of processes and data flows). Even intermediate versions are not very representative as the final version contains more than double the amount of data flows. This observation further stresses the need of continually revisiting the DFD model as an application is further developed.

Finally, there is another type of change not visible in diagrams such as Fig. 7.6. These involve changes where elements are replaced or when some elements are deleted at the same time as other elements are added. These changes do not impact the model sizes but do result in different models with different threats.

#### 7.5.2.2 Threat Analysis Results

Next, the results of the threat elicitation and risk analysis are discussed. Figures 7.7 and 7.8 visualize the evolution of, respectively, the number of threats and the (inherent and residual) risk. The evolution of the number of threats (Fig. 7.7) again confirms the importance of frequently revisiting the threat model: many new threats are introduced over time as the system is extended with new functionality. This graph also shows that even minor version changes (e.g., `v10`) can introduce a significant number of new threats.

The graph of the inherent and residual risk (Fig. 7.8) shows a similar increasing trend over time. But the risk also incorporates the effects of (partially) mitigated threats as these reduce the residual risk. On this graph, similar patterns of evolution of the inherent and residual risk can be discerned as for the artificial changes used

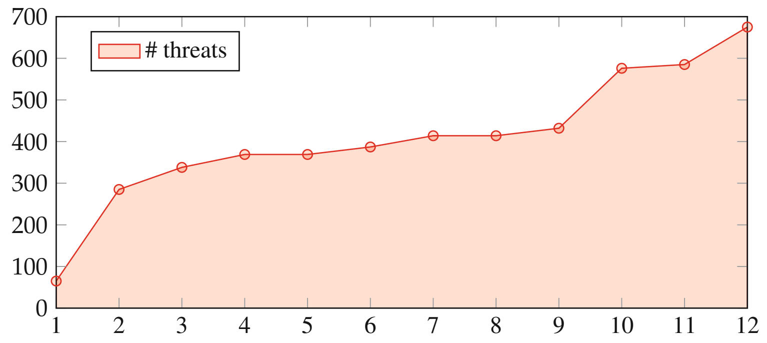


**Fig. 7.7** Overview of the threat evolution. This plot shows how the number of threats evolves with each new version as the system is extended and modified

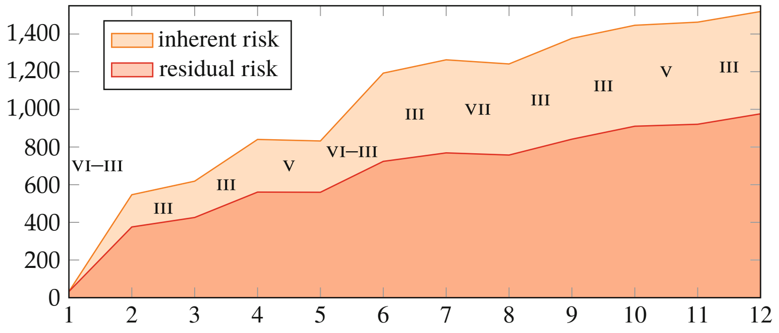


**Fig. 7.8** Overview of the analysis results for the different changes. This plot shows the resulting inherent and residual risk values over time and the corresponding patterns from Table 7.1

in Sect. 7.5. On Fig. 7.8, each part of the graph between two commits is annotated with the corresponding pattern from Table 7.1. These evolution patterns are often combinations of two patterns (e.g., 1–2 and 5–6 are obvious combinations and marked in Fig. 7.8), as the changes to a project over multiple versions rarely consist exclusively of either functionality- or security-related modifications. Some are more subtle, as many of the pattern III instances will not have perfectly parallel lines. Finally, note that evolution pattern III ("add insecure functionality") is the most frequent pattern in Fig. 7.8. This is not due to the application being insecure, but because our models only include a subset of the solutions that are used in the actual application. Hence, the occurrence of this pattern may also serve as an indicator for discrepancies between the model (or, more specifically, the modeled solutions) and the actual application (see Sect. 7.6.5).

## 7.6 Discussion

This section discusses several considerations and limitations of using CTAM in practice. Ideas for further improvements are not discussed here but are deferred to Sect. 7.7.

### *7.6.1 Using Another Threat Elicitation Engine*

As discussed in Sect. 7.3.2, the presented prototype is built around the threat elicitation engine of SPARTA [45] because of its powerful enablers. Any alternative threat modeling tool could in theory be adopted, provided that it generates a list of threats, with, for every threat, (i) the threat type, (ii) the affected DFD model element, (iii) the data flow, and (iv) the estimates of the inherent risk (i.e., the risk when ignoring all countermeasures) and the residual risk (i.e., taking into account security and privacy solutions) For example, CTAM could be integrated with Pytm [51] to elicit threats, provided that Pytm is extended with (i) a risk estimation approach, such as FAIR [15], or (more pragmatically) a translation of its current severity categories to numerical values and (ii) the possibility to elicit threats that have been mitigated by a solution to ensure that these can be taken into account when tracking the progress that is being made.

### *7.6.2 Scope of the Model*

Another attention point is the scope of the DFD model and the resulting threat model. When the DFD model is included in a repository that also contains source code artifacts, it should at least represent that application. However, this application will, in practice, be deployed and interact with other entities (such as other applications, end users, or third-party services) that do not necessarily have a representation in the source code. This raises the issue on which entities should be encoded in the DFD model.

The most straightforward approach entails the analysis of a single application and its internal security properties. Yet the usage of threat analysis as part of a CI pipeline does not prevent the use of a broader scope of the analysis. Indeed, the use of infrastructure as code [34] makes operational information available as source code artifacts that can also be used as inputs for the models. Furthermore, submodules can be used to combine multiple related repositories (i.e., components of a larger application), where the DFD model resides in the root repository and refers to the application as a whole. This approach was taken for the contact tracing application in Sect. 7.5, where multiple (microservice) repositories are combined as submodules in a single repository that contains the overall threat model.

### *7.6.3 Model Granularity*

Compared to source code, a DFD depicts a more abstract view of a software system. There is, however, a degree freedom in the abstraction level of a DFD: its elements could denote running processes, classes, methods, and so forth. Choosing and managing the abstraction level of the DFD, and thus the abstraction gap between the DFD and the source code, require careful consideration when applying automated threat modeling.

If the abstraction gap is small and the DFD closely resembles the source code, threat analysis results will include detailed information, and elicited threats can be more easily linked back to the source code. However, code changes are more likely to warrant model updates, so more effort is required to keep the model up to date with the source code (as will be discussed in Sect. 7.6.5). Moreover, the DFDs may become large and lead to many elicited threats.

This is not the case if the DFD depicts a higher-level overview of the system and is less closely linked to the source code. In this case, however, analysis results will also be high level, thus requiring more efforts to link threats back to source code and mitigate them.

For our evaluation of the contact tracing application, we chose a level of granularity that was primarily driven by the annotations of the Spring framework, as discussed in Sect. 7.5.

### *7.6.4 Triggering the Analysis Process*

CTAM analyzes the DFD models from the repository on every individual git commit. However, individual git commits may not be the appropriate level of input granularity for threat analysis and monitoring over time. We briefly discuss the trade-off between analyzing individual commits and using alternative triggers (such as pull or merge requests).

The analysis of the threat model at the granularity of individual commits introduces additional overhead of maintaining and updating the DFD model for every commit that introduces relevant changes. This approach makes it trivial to link newly discovered security or privacy threats to the relevant source code portions that introduced the threat, because the relevant source code and threat model changes belong to the same commit. A variation on this approach is reanalyzing the model on every commit in which the model changed. When the source code and model are not updated as part of the same commit but assuming that the model is eventually made consistent with the source code, then the code that introduced a threat can be found between the commits of the model that first generated that threat and the previous model.

An alternative approach is to rely on different triggers for re-evaluating the threat model. Alternative triggers could be specific milestones (or git tags) or when merge or pull requests are submitted to the repository. Such an approach could be used to enforce practices such as requiring an assessment of the impact on the threat modeling before merging new feature branches. The potential downside of these more coarse-grained approaches is that there is no longer a direct connection between the model and the relevant source code changes. This disconnect makes it more difficult to identify the relevant source code portions requiring review to mitigate newly introduced security or privacy threats or remove the insecure functionality altogether.

### *7.6.5 Avoiding Model Drift*

CTAM currently relies on the inclusion of a DFD model in the code base that is kept up to date throughout the development. In case this model deviates (e.g., as a consequence of architectural drift [52]) from reality, the usefulness of the presented approach decreases drastically, as not all the generated threats will be relevant (false positives) or not all the relevant threats will be identified (false negatives). Additionally, the modifications in a single commit may not always necessitate changes to the model itself, as this depends on the granularity of the commits. There are opportunities, however, to systematically revisit the accuracy of the model as part of, for example, merge requests that introduce more considerable changes.

While the above argument applies to any threat modeling approach, the integration of threat analysis activities into the code versioning system presents two opportunities for improvements in this regard: First, techniques for architectural reconstruction and conformance checking can be used to validate the accuracy of the input model vis-a-vis the committed code. Second, the use of code annotations for the construction of the input model can remove the need for a separate centralized input model altogether. These options are further discussed in more detail in Sect. 7.7.

### *7.6.6 Using Detailed Threat Analysis Information*

As demonstrated in Sect. 7.5, CTAM provides immediate feedback on the progress being made in creating a secure- and privacy-preserving design in terms of the *inherent risk* and the *residual risk* which are both aggregated (by addition). These values are calculated and reported for each commit.

This degree of integration with version control systems allows for a number of additional interesting analyses on the evolution of a code base. For example, proposed changes in other branches or merge/pull requests could be analyzed and compared with the main branch to evaluate the security and privacy impact the merge would have on the main branch.

Because SPARTA performs a fine-grained risk assessment, more detailed intermediary risk analysis results can be used (e.g., the effectiveness of specific solutions or the impact on specific data subject types) instead of the aggregated risk estimates per threat. This would allow the developer to perform more targeted assessments, e.g., the analysis of privacy risk from the perspective of a specific data subject type and its evolution over time, or focused on specific assets (e.g., credit card numbers or user data), or filtering on specific model elements.

### *7.6.7 Security Metrics*

The systematic analysis and measurement of a software product necessarily bring us to the domain of software security metrics: a difficult, if not infeasible [2], endeavor. Despite the inherent difficulties, many proposals have been made in the literature to measure different security-relevant properties, such as dependency graphs [30], attack surfaces [25], and software metrics [23, 26, 27]. While the risk assessments of the elicited threats may not be suitable as a metric to compare the security of different software products in absolute terms, it does allow monitoring the progress that is being made in securing one specific system throughout its development. For example, the difference between the inherent and residual risk, and the evolution thereof, can already serve as a crude indicator for the degree of security of the application's design. Furthermore, our prototype lays the groundwork and provides a generic framework for future evaluation of, and experimentation with, calculating and comparing different security or privacy metrics over time.

## 7.7 Future Work

This section outlines our roadmap of future work grouped in terms of (i) how to acquire or maintain DFD model inputs for the analysis, (ii) alignment with operational analysis and monitoring, and (iii) the meta-analysis over multiple historical versions of a software development project.

### *7.7.1 DFD Model Inputs*

In the current prototype, CTAM relies upon the presence and maintenance of up-to-date DFD models that are encoded as textual files (e.g., in CTAM, these are `.sparta` DFD files) and thus can be checked in along with regular code commits, in branches, etc. The main idea is that the developer manually updates and coevolves these models alongside his regular code update, whenever these warrant a change to the DFD itself.

This is however suboptimal, as (i) it requires continuous attention from the developer, (ii) and as such, it easily becomes an oversight, and (iii) there are no guarantees the DFD model is and remains consistent with the actual code base as it stands per code revision.

In this area, model-centric techniques such as automated architecture extraction [4, 57] or reconstruction [16, 42, 59], software reflexion models [5, 29], static model compliance checking [24, 33], and model coevolution [17, 28] are particularly promising. Their further implementation and integration are therefore considered future work.

An interesting subproblem and trade-off are related to the possible divergence that may occur when such a model is coevolved gradually and incrementally with each revision, on the one hand, and the cost and limited scalability of approaches that reconstruct a system model from scratch for each revision, on the other hand. Here, the ability to construct DFD models from individual model fragments (e.g., through process nesting as it is currently supported in SPARTA or through composition of DFDs corresponding to individual sub-projects or modules) may present an additional opportunity for improvement and optimization.

In each of these approaches, automation of such extraction methods is a key requirement to align with the vision of automated and integrated threat analysis presented in this article. Methods that require manual interventions by the developer or threat modeler are considered suboptimal in this context.

Nevertheless, due to the challenge of automatically deriving a useful DFD from the source code, it is worthwhile to explore code-oriented threat modeling tools, such as ThreatSpec [53]. These rely on code annotations (manually added by the developer) for the construction of the input model and can remove the need for a separate centralized input model altogether.

### *7.7.2 Monitoring and Aligning the Operational System*

The DFD models used by CTAM predominantly maintain an architecture- and development-centric perspective on the system. In addition to the challenges inherent to aligning these models with development artifacts discussed in the previous section, additional relevant inputs may be considered that come from the operational context, which is also a valid and accessible source of information in a DevOps continuous integration pipeline.

For example, run-time inspection and monitoring techniques may be used to check deviation between the run-time system and the system model encoded in the DFD, an activity that is called conformance checking [14] or the extraction of architectural models at the basis of run-time interactions [4, 21, 57]. Operational security technologies could inform and update the threat model about detected attacks, anomalies, or changes to the expected workload, which may indicate deviations between the system design and the DFD or inform the security solutions modeled within the DFD in terms of strength or coverage of risk.

In addition, run-time security adaptive mechanisms such as run-time application self-protection (RASP) are capable of changing the fundamental structure of a system in response to a security issue, and these changes must also be reflected in the DFD to allow the CTAM risk analysis to take into account such risk mitigations enacted dynamically.

The challenges and benefits to the holistic ingestion of these diverse sources of information coming from a modern DevOps environment have been discussed in earlier work [55]. Additional work to further explore and validate these promising integrations of different sources of operational information is ongoing.

### *7.7.3 Project-Centric Risk Analysis and Management Use Cases*

The use of CTAM promotes the creation and coevolution of threat models for each intermediate version and at each snapshot of the development of a system. Next to the system models (DFDs), the outcome of threat analysis – lists of threat scenarios, ranked in accordance with their priority/risk – constitutes a knowledge repository that can be monitored and mined for valuable insights about the evolution and management of risk in a project over time.

A number of promising novel use cases can be discussed in this regard. For example, this approach would allow (i) monitoring the recurrence or re-emergence of threats over time which indicates possible regressions, (ii) observing an evolution in the types of threats raised which may be indicative of the overall project evolution (e.g., the project evolves to rely more extensively on personal data and thus privacy threats will be raised more), (iii) alternative risk aggregation functions for monitoring, (iv) the overall increase of inherent risk with the introduction of new features, (v) the ability of existing solutions or countermeasures to reduce or manage some of the newly emerging risks (i.e., the effectiveness of countermeasures, the stability of the security architecture over time, etc.), and (vi) security solutions or mitigations that can be suggested for specific threats and risk sources in the system, and they can be evaluated at the basis of their risk reduction outcomes. In these use cases, the security architect, project manager, software developer, and operator can be provided with more direct feedback and actionable insights that may allow them to further optimize the development process.

Performing these types of analysis activities and validating the different analysis scenarios discussed above in a real-world case are considered part of our future work.

## 7.8 Conclusion

Threat analysis is commonly performed in a single-shot operation in the early stages of software development. Because of this, progress in threat mitigation is not actively revisited and monitored throughout later development stages such as the implementation and as the system evolves over time. Furthermore, as changes are made to the system, the originally anticipated threats may become obsolete while novel threats remain undiscovered.

In this chapter, we have introduced CTAM, a continuous threat analysis and management prototype that supports continuous threat modeling and elicitation and integrates this activity into a continuous integration pipeline in GitLab. By revisiting threat analysis as new changes are pushed to the source code repository, threat management becomes a continuous activity, and the progress in mitigating threats (both in applying appropriate security and privacy solutions as in making changes to existing functionality) can be more accurately monitored.

Integrating threat analysis activities in a continuous integration pipeline provides the following benefits: First, threat management becomes a continuous concern, rather than a single-shot analysis on an outdated version of the system. Second, it provides guidance toward mitigating threats and keeps track of the progress. Third, it creates the need to maintain the architectural abstraction model of the system and forces developers to reflect on the broader architectural impact of their changes in terms of security and privacy.

**Acknowledgments** We thank Robin Vanhove for his contributions in the creation of the DFD models as part of his master's thesis. This research is partially funded by the Cybersecurity Initiative Flanders and the KU Leuven research fund.

## References

1. Audodesk: Autodesk Continuous Threat Modeling (2021). https://github.com/Autodesk/continuous-threat-modeling/
2. S. Bellovin, On the brittleness of software and the infeasibility of security metrics. IEEE Secur. Priv. **4**(4), 96–96 (2006). https://doi.org/10.1109/MSP.2006.101
3. Z. Braiterman, A. Shostack, J. Marcil, S. de de Vries, I. Michlin, K. Wuyts, R. Hurlbut, B.S. Schoenfield, F. Scott, M. Coles, C. Romeo, A. Miller, I. Tarandach, A. Douglen, M. French, *Threat Modeling Manifesto* (2020). https://www.threatmodelingmanifesto.org/
4. F. Brosig, N. Huber, S. Kounev, Automated extraction of architecture-level performance models of distributed component-based systems, in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)* (IEEE, 2011), pp. 183–192
5. J. Buckley, S. Mooney, J. Rosik, N. Ali, JITTAC: a just-in-time tool for architectural consistency, in *2013 35th International Conference on Software Engineering (ICSE)* (2013), pp. 1291–1294. https://doi.org/10.1109/ICSE.2013.6606700
6. Corona-Warn-App: Corona-Warn-App: Documentation (2022). https://github.com/corona-warn-app/cwa-documentation

7. Corona-Warn-App: Corona-Warn-App server (2022). https://github.com/corona-warn-app/cwa-server
8. Corona-Warn-App: Corona-Warn-App testresult server (2022). https://github.com/corona-warn-app/cwa-testresult-server
9. Corona-Warn-App: Corona-Warn-App verification iam (2022). https://github.com/corona-warn-app/cwa-verification-iam
10. Corona-Warn-App: Corona-Warn-App verification portal (2022). https://github.com/corona-warn-app/cwa-verification-portal
11. Corona-Warn-App: Corona-Warn-App verification server (2022). https://github.com/corona-warn-app/cwa-verification-server
12. S.T. Datko, Static code analysis with GitLab-CI. Tech. rep. (2016)
13. M. Deng, K. Wuyts, R. Scandariato, B. Preneel, W. Joosen, A privacy threat analysis framework: supporting the elicitation and fulfillment of privacy requirements. Requir. Eng. **16**(1), 3–32 (2011)
14. S. Dunzer, M. Stierle, M. Matzner, S. Baier, Conformance checking: a state-of-the-art literature review, in *Proceedings of the 11th International Conference on Subject-Oriented Business Process Management*, 2019, pp. 1–10
15. J. Freund, J. Jones, *Measuring and Managing Information Risk: A FAIR Approach* (Butterworth-Heinemann, 2014)
16. J. Garcia, I. Ivkovic, N. Medvidovic, A comparative analysis of software architecture recovery techniques, in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (IEEE, 2013), pp. 486–496
17. S. Getir, A. Van Hoorn, L. Grunske, M. Tichy, Co-evolution of software architecture and fault tree models: an explorative case study on a pick and place factory automation system. NiM-ALP@ MoDELS **13**, 32–40 (2013)
18. GitLab: GitLab CI/CD (2021). https://docs.gitlab.com/ee/ci/
19. IriusRisk: IriusRisk (2021). https://www.iriusrisk.com/
20. O. Javed, J.H. Dawes, M. Han, G. Franzoni, A. Pfeiffer, G. Reger, W. Binder, PerfCI: a toolchain for automated performance testing during continuous integration of Python projects, in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (IEEE, 2020), pp. 1344–1348
21. M. Kleehaus, Ö. Uludağ, P. Schäfer, F. Matthes, Microlyze: a framework for recovering the software architecture in microservice-based environments, in *International Conference on Advanced Information Systems Engineering* (Springer, 2018), pp. 148–162
22. R. Kozik, M. Choraś, D. Puchalski, R. Renk, Platform for software quality and dependability data analysis, in *International Conference on Dependability and Complex Systems* (Springer, 2018), pp. 306–315
23. M. Lanza, S. Ducasse, Understanding software evolution using a combination of software visualization and software metrics, in *Proceedings of LMO 2002 (Langages et Modèles à Objets* (Lavoisier, 2002), pp. 135–149
24. F. Leymann, V. Yussupov, U. Zdun, Monitoring behavioral compliance with architectural patterns based on complex event processing, in *Service-Oriented and Cloud Computing: 8th IFIP WG 2.14 European Conference, ESOCC 2020*, Heraklion, Crete, Greece, 28–30 Sept 2020, Proceedings, vol. 12054 (Springer Nature, 2020), p. 125
25. P.K. Manadhata, J.M. Wing, An attack surface metric. IEEE Trans. Softw. Eng. **37**(3), 371–386 (2011). https://doi.org/10.1109/TSE.2010.60
26. N. Medeiros, N. Ivaki, P. Costa, M. Vieira, Software metrics as indicators of security vulnerabilities, in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, Oct 2017, pp. 216–227. https://doi.org/10.1109/ISSRE.2017.11
27. T. Mens, S. Demeyer, Future trends in software evolution metrics, in *Proceedings of the 4th International Workshop on Principles of Software Evolution. IWPSE '01* (Association for Computing Machinery, New York, NY, USA, 2001), pp. 83–86. https://doi.org/10.1145/602461.602476

28. R. Morrison, D. Balasubramaniam, F. Oquendo, B. Warboys, R.M. Greenwood, An active architecture approach to dynamic systems co-evolution, in *European Conference on Software Architecture* (Springer, 2007), pp. 2–10
29. G. Murphy, D. Notkin, K. Sullivan, Software Reflexion models: bridging the gap between design and implementation. IEEE Trans. Softw. Eng. **27**, 364–380 (05 2001). https://doi.org/10.1109/32.917525
30. V.H. Nguyen, L.M.S. Tran, Predicting vulnerable software components with dependency graphs, in *Proceedings of the 6th International Workshop on Security Measurements and Metrics. MetriSec '10* (Association for Computing Machinery, New York, NY, USA, 2010). https://doi.org/10.1145/1853919.1853923
31. OWASP: OWASP top 10 – 2021 (2021). https://owasp.org/Top10/
32. OWASP: Threat Dragon (2021). https://owasp.org/www-project-threat-dragon/
33. S. Peldszus, K. Tuma, D. Strüber, J. Jürjens, R. Scandariato, Secure data-flow compliance checks between models and code based on automated mappings, in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)* (2019), pp. 23–33. https://doi.org/10.1109/MODELS.2019.00-18
34. A. Rahman, R. Mahdavi-Hezaveh, L. Williams, A systematic mapping study of infrastructure as code research. Inf. Softw. Technol. **108**, 65–77 (2019). https://doi.org/https://doi.org/10.1016/j.infsof.2018.12.004, https://www.sciencedirect.com/science/article/pii/S0950584918302507
35. T. Rangnau, R.V. Buijtenen, F. Fransen, F. Turkmen, Continuous security testing: a case study on integrating dynamic security testing tools in ci/cd pipelines, in: *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)* (2020), pp. 145–154. https://doi.org/10.1109/EDOC49727.2020.00026
36. K. Rindell, K. Bernsmed, M.G. Jaatun, Managing security in software – or: how i learned to stop worrying and manage the security technical debt, in *Proceedings of the 14th International Conference on Availability, Reliability and Security (ARES'19)* (ACM, 2019), pp. 1–8. https://doi.org/10.1145/3339252.3340338
37. M. Rusakovich, TicTaaC (2021). https://github.com/rusakovichma/TicTaaC
38. C. Schneider, Threagile (2021). https://threagile.io/
39. A. Shostack, Experiences threat modeling at Microsoft, in *Modeling Security Workshop* (Dept. of Computing, Lancaster University, UK, 2008)
40. A. Shostack, *Threat Modeling: Designing for Security* (John Wiley & Sons, Indianapolis, Indiana, 2014)
41. G.B. Simpson, CI/CD Software Security Automation. Tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States) (2018)
42. Z.T. Sinkala, M. Blom, S. Herold, A mapping study of software architecture recovery for software product lines, in *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings* (2018), pp. 1–7
43. L. Sion, D. Van Landuyt, W. Joosen, The never-ending story: On the need for continuous privacy impact assessment, in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)* (IEEE, 2020), pp. 314–317
44. L. Sion, D. Van Landuyt, K. Wuyts, W. Joosen, Privacy risk assessment for data subject-aware threat modeling, in *2019 IEEE Security and Privacy Workshops (SPW)* (IEEE, 2019)
45. L. Sion, D. Van Landuyt, K. Yskout, W. Joosen, SPARTA: Security & privacy architecture through risk-driven threat assessment, in *IEEE 2018 International Conference on Software Architecture (ICSA 2018)* (IEEE, 2018). [freely]
46. L. Sion, D. Van Landuyt, K. Yskout, S. Verreydt, W. Joosen, Automated threat analysis and management in a continuous integration pipeline, in *2021 IEEE Secure Development Conference (SecDev)* (IEEE, 2021), pp. 30–37
47. L. Sion, K. Wuyts, K. Yskout, D. Van Landuyt, W. Joosen, Interaction-based Privacy threat elicitation, in *Proceedings of the 4th International Workshop on Privacy Engineering – IWPE 2018* (IEEE, 2018)

48. L. Sion, K. Yskout, D. Van Landuyt, W. Joosen, Risk-based Design security analysis, in *Proceedings – 2018 IEEE/ACM First International Workshop on Security Awareness from Design to Deployment, SEAD 2018*, Gothenburg, Sweden (2018)
49. L. Sion, K. Yskout, D. Van Landuyt, W. Joosen, Solution-aware data flow diagrams for security threat modelling, in *Proceedings of The 6th Track on Software Architecture: Theory, Technology, and Applications* (2018)
50. K. Tan, V. Garg, An analysis of open-source automated threat modeling tools and their extensibility from security into privacy. Usenix; login: online publication (2022). https://www.usenix.org/publications/loginonline/analysis-open-source-automated-threat-modeling-tools-and-their
51. I. Tarandach, Pytm (2021). https://github.com/izar/pytm
52. B. Tekinerdogan, Architectural drift analysis using architecture reflexion viewpoint and design structure reflexion matrices, in *Software Quality Assurance* (Elsevier, 2016), pp. 221–236
53. ThreatSpec: ThreatSpec (2021). https://threatspec.org/
54. K. Tuma, C. Sandberg, U. Thorsson, M. Widman, T. Herpel, R. Scandariato, Finding security threats that matter: two industrial case studies. J. Syst. Softw. 111003 (2021). https://doi.org/10.1016/j.jss.2021.111003
55. D. Van Landuyt, L. Pasquale, L. Sion, W. Joosen, Threat models at run time: the case for reflective and adaptive threat management (nier track) (2021)
56. C. Vassallo, S. Proksch, A. Jancso, H.C. Gall, M. Di Penta, Configuration smells in continuous delivery pipelines: a linter and a six-month study on GitLab, in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2020), pp. 327–337
57. M. Walter, R. Heinrich, R. Reussner, Architectural attack propagation analysis for identifying confidentiality issues, in *2022 IEEE 19th International Conference on Software Architecture (ICSA)* (IEEE, 2022), pp. 1–12
58. K. Wuyts, Privacy Threats in Software Architectures. PhD Thesis, KU Leuven (2015)
59. T. Yang, Z. Jiang, Y. Shang, M. Norouzi, Systematic review on next-generation web-based software architecture clustering models. Comput. Commun. **167**, 63–74 (2021)
60. K. Yskout, T. Heyman, D. Van Landuyt, L. Sion, K. Wuyts, W. Joosen, Threat modeling: from infancy to maturity, in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results* (ACM, 2020), pp. 9–12. https://doi.org/10.1145/3377816.3381741

# Chapter 8
# EARLY: A Tool for Real-Time Security Attack Detection

**Tanwir Ahmad, Dragos Truscan, and Jüri Vain**

**Abstract** The Internet has become a prime subject of security attacks and intrusions by attackers. These attacks can lead to system malfunction, network breakdown, data corruption, theft, etc. A network intrusion detection system (IDS) is a tool used for identifying unauthorized and malicious behavior by observing network traffic. State-of-the-art IDSs are designed to detect an attack by inspecting the complete information about the attack. This means that an IDS would only be able to detect an attack after it has been executed on the system under attack and might have caused damage to the system. In this paper, we extend our early IDS proposed in our previous work. The tool can detect network attacks before they could cause any more damage to the system under attack while preventing unforeseen downtime and interruption. In this work, we employ different deep neural network architectures for attack identification and compare their performances. The deep neural networks are trained in a supervised manner to extract relevant features from raw network traffic data instead of relying on a manual feature selection process used in most related approaches. Further, we empirically evaluate our tool on two datasets from different domains: CICIDS2017 from the web application domain and MQTT-IDS-2020 dataset from the IoT domain. The results show that our approach performed well and attained a high overall balanced accuracy.

**Keywords** Convolutional neural network · Gated recurrent unit · Intrusion detection system · Early detection

T. Ahmad · D. Truscan (✉)
Department of Information Technology, Åbo Akademi University, Turku, Finland
e-mail: tanwir.ahmad@abo.fi; dragos.truscan@abo.fi

J. Vain
High-Assurance Software Laboratory, Tallinn University of Technology, Tallinn, Estonia
e-mail: juri.vain@ttu.ee

A. Sadovykh et al. (eds.), *CyberSecurity in a DevOps Environment*,
https://doi.org/10.1007/978-3-031-42212-6_8

## 8.1 Introduction

Modern society is significantly dependent on a wide range of interconnected software systems for finance, energy distribution, communication, and transportation. The era of controlled communication in closed networks for restricted purposes is over. Due to the adoption of Internet technologies, almost all financial, government, and social sectors started to rely heavily on networked information systems to process and store confidential information. As a result, these systems have become primary subjects to security attacks and intrusions by attackers. These attacks can lead to system malfunction, network breakdown, data corruption, theft, etc. Therefore, it is essential to ensure network security by monitoring and detecting network attacks in real time as early as possible.

A network IDS is a tool used for identifying unauthorized and malicious behavior by observing the network traffic and helping network administrators take appropriate reactive measures to secure the network infrastructure and the associated nodes [29]. The majority of the IDSs can be divided into two groups: *anomaly-based* and *signature-based* detection systems [4]. In the former group, a detection system learns the profile of normal network traffic and would classify the given network traffic data as intrusive or anomalous if it deviates from the normal traffic profile by more than a predefined anomaly threshold. This allows these systems to detect undiscovered and novel attacks. However, the value of the anomaly threshold has a significant impact on the accuracy of the systems. Finding the optimal value of the anomaly threshold is a complicated task and, in many cases, requires manual tuning.

A *signature-based* IDS identifies intrusive network traffic by comparing the given network traffic data against the signatures (e.g., sequence of string and regular expressions) of known attacks. This category of network IDS is most commonly used in daily practice [38]. Since most of these systems rely on the knowledge bases (i.e., predefined sets of attack models and patterns) extracted from known attacks and system vulnerabilities, they are also known as *knowledge-based* or *misuse* IDS [22]. In most cases, the domain experts construct the knowledge bases manually, which can be a tedious and error-prone task [21]. Unlike anomaly-based IDSs, this group of IDSs can only detect those attacks that are defined in the knowledge bases. However, these methods demonstrate a high degree of accuracy and a low false alarm rate compared to the anomaly-based IDSs [4, 12]. One of the main challenges in developing these systems is extracting or defining a signature of a known attack that can represent different variations of the attack. Furthermore, managing a large signature knowledge base and matching signatures against the traffic are time- and resource-intensive tasks [4].

The spread of high-speed networks and fast-propagating threats poses additional challenges to current IDSs, which detect an attack by inspecting the entire network traffic data related to the attack. This means that an IDS would only be able to detect an attack after it has been executed on the system under attack and might have already caused damage to the system. Therefore, early attack (or intrusion) detection is desirable in the cybersecurity domain to prevent network attacks before they could cause any more damage to the system. Based on the early classification results of the traffic, the network administrator can decide whether to stop the traffic and raise an alarm message or deploy countermeasures.

In order to address the above challenges, in our previous work [3], we proposed an end-to-end signature-based early IDS to detect network attacks as early as possible with a high degree of accuracy. To that extent, we employed a one-dimensional convolutional neural network (1D CNN) [20] (i.e., a type of *deep neural network* (DNN) [7]) to extract relevant features from raw network traffic data that were used for early classification of ongoing attacks. Deep learning is a type of machine learning where we utilize DNNs [7] or multi-layer neural networks to approximate complex functions by learning different levels of representations of the given training data. In order to evaluate the performance of our tool, we have also defined a new metric, *earliness*, which measures the ratio of information of an attack needed to classify it as a given attack type.

In this work, we extend our previous work by employing and comparing different types of neural networks for early detection of network attacks. Further, we rigorously evaluate the applicability of our approach by benchmarking its performance on two datasets from different domains: CICIDS2017 dataset [33] from the web application domain and MQTT-IDS-2020 [16] dataset from the IoT domain.

In summary, the contributions of this work are as follows:

1. We present a tool-supported network IDS focused on optimizing the accuracy of attack detection with minimum feasible delay (or, in other words, maximum earliness).
2. The tool has two main components: one for creating attack detection models for different application domains and one for monitoring the network traffic with respect to the type of attack model chosen.
3. The attack model creation component automatically extracts the relevant features from raw network traffic data in an end-to-end manner instead of relying on the manual feature engineering process. Therefore, our approach is domain-independent and does not require domain-specific data preprocessing steps.
4. The monitoring component works at the network packet level, and thus it is agnostic to the type of protocol being monitored. In addition, it allows one to select which type of attacks is being monitored based on the type of the selected attack model.
5. Two neural network architecture types can be selected, RNN and CNN, depending upon the need for real-time detection time delay, precision, or training time.
6. We introduce a new metric, called *earliness*, to evaluate how early our tool can detect attacks.
7. We empirically evaluate the detection capability and earliness of the tool using several datasets in two application domains.

## 8.2 Overview of the Early Tool

The main goal of the EARLY tool is to monitor the network traffic in real time against known security attacks for different application domains and deploy countermeasures before the attacks are completed. The main feature of this tool is to detect ongoing attacks with high accuracy.

The EARLY tool can be integrated with DevOps environments allowing an organization to continuously monitor its systems and identify potential security threats before they escalate into larger-scale incidents. The tool has three main components: a training module, a library of trained models, and a monitoring module (see Fig. 8.1). The *training module* is used during the *Develop* and *Release* phases to train neural network models using various datasets for different application domains based on decisions taken in the *Plan* phase. The resulting model is saved in the *library of attack models* and used later on by the *monitoring module* to monitor the corresponding type of network traffic during the *Monitor* phase. Whenever attacks are detected, automatic countermeasures can be deployed based on predefined triggers.

The tool works at the network packet level, by analyzing network flows. A *network flow* is a bidirectional sequence of packets exchanged between two endpoints (e.g., a web server and a client) during a certain time interval with some common flow properties [9] such as source and destination IP addresses, source and destination port numbers, and the protocol type. In our work, we define a network flow as a sequence of $T$ ordered packets, where $T$ represents the length of a complete flow. A flow is denoted as:

$$F_T = \{P_1, P_2, \ldots, P_T\}, \ \forall \ P_i \in \mathbb{R}^d \wedge 1 \leq i \leq T \tag{8.1}$$

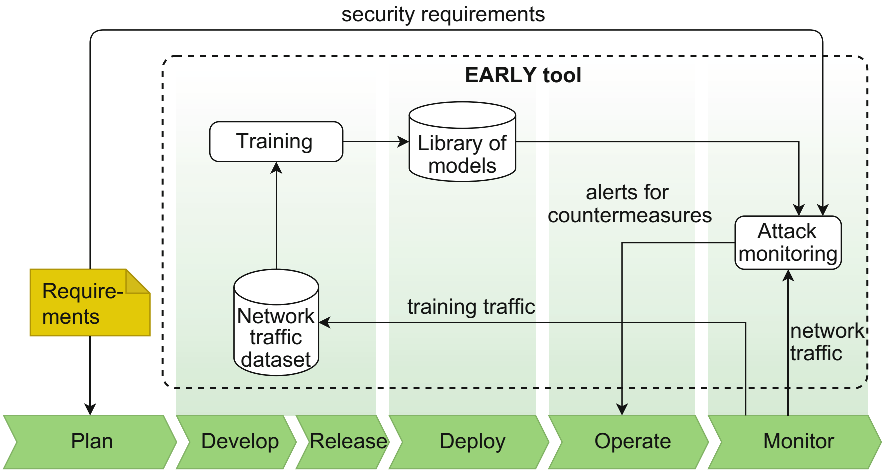


**Fig. 8.1** Overview of EARLY tool architecture in the DevOps context

where *d* is the dimension (or length) of a packet.

In order to extract flows from the network traffic, both the training module and the monitoring module use a sub-module, called **flow processing**.

## 8.2.1 Flow Processing

The *flow processing* module consists of three components: packet filtering, flow identification, and packet preprocessing module, as shown in Fig. 8.2.

### 8.2.1.1 Packet Filtering

The packet filtering component captures the network packets and forwards them to the subsequent components if they satisfy the given criteria. For example, if we are protecting the web server running at port 80, we can configure the component to forward only those packets whose destination or source port is 80. We monitor the raw network traffic between the system under attack and the untrusted network. We select only those network packets which are related to the type of attacks we would like to detect. For example, if we are interested in detecting only web attacks [17], we will capture only HTTP packets.

The next two modules transform the packets and group them into network flows. Whenever a network flow is updated with a new packet, we use the early flow classifier to update the prediction corresponding to the flow.

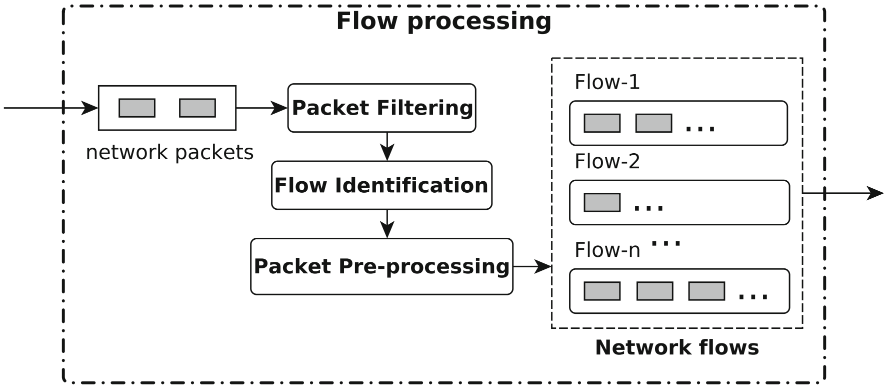


**Fig. 8.2** Flow processing module

#### 8.2.1.2 Flow Identification

Upon receiving a new packet, we inspect the packet properties such as source and destination IP addresses to identify a suitable active flow for it. An *active flow* represents an ongoing communication session between a pair of network endpoints. On the other hand, if we cannot find an active flow that matches the characteristics of a packet, we create a new flow. A network flow is considered to be *terminated* or *inactive* upon connection teardown (e.g., by FIN packet) or when the flow has not received a new packet within a certain flow time-out (e.g., 120 seconds). The flow time-out value can be adjusted according to the protocol type of the network traffic we are capturing for detecting attacks.

#### 8.2.1.3 Packet Preprocessing

Once we have identified the appropriate flow for the new packets, each packet goes through the following preprocessing steps to truncate unwanted information and transform it to a uniform-size vector of bytes (truncation and transformation operations). The main purpose of the steps is to ensure that the classifier should rely on relevant features for flow classification. For exemplification purposes, in the following, we discuss the steps in relation to HTTP and TCP protocol; however, these steps can be applied to other types of network packets with minor modifications.

**Truncation** removes irrelevant headers and fields so that the classifier focuses only on useful features for flow classification and does not over-fit unrelated features such as MAC and IP addresses.

For instance, the raw captured packets contain the Ethernet header. The header has information concerning the physical link, such as the media access control (MAC) address used for transferring the frames between different nodes in the network. However, this information is valueless for attack identification because it can be spoofed easily. Thus, this header is removed from the packet.

Similarly, the Internet Protocol (IP) header in the packets includes information such as the total length of the packet, protocol version, and source and destination IP addresses. This information is necessary for routing packets in the network. However, we consider this information irrelevant and counterproductive for our classifier since there is a chance that the classifier will start relying on the IP information (e.g., IP addresses) for detecting attack flows. Therefore, we remove it from the packets. This approach allows the classifier to function steadily even if the addresses of the nodes in the network have changed and generalize the knowledge learned from one network environment to another.

**Transformation** A fixed-size input is required when using a neural network for classification. To make the length of the header of the transport layer and the payload of the packets uniform, we crop or pad them with zeros to a fixed length. We would like to point out that even though we restrict the length of the packets (i.e., $d$ in

Eq. 8.1) in a flow, we do not restrict the length of a flow (i.e., number of packets $T$) unlike other proposed approaches (e.g., [44]) though it is implicitly bounded by time-out.

### *8.2.2 Training*

We train the classifier offline before using it for online early IDS. The training process is depicted in Fig. 8.3. We require a labeled flow dataset for supervised training that contains normal and attack flows. In addition to the labeled flows, the dataset should also have network packets corresponding to the flows.

The majority of the publicly available datasets used for training and evaluating the IDSs have the class imbalance problem [4], that is, the number of examples among the different classes is not similar in the dataset. A classifier trained on an imbalanced dataset typically exhibits poor performance in terms of overall prediction accuracy.

Therefore, in this work, in order to rectify the effect of class imbalance, we use a cost-sensitive learning method [15]. In this method, we train the neural network model with sample weighting, which acts as a coefficient for the loss value computed for each sample (i.e., flow) during the training process. The weight of each sample is based on its class. It is calculated inversely proportional to the class frequencies in the training data. The objective is that the classifier should pay more attention to those samples that belong to an underrepresented class.

Previous work [27, 36] has shown that the cost-sensitive learning method performs better than the sampling-based method (such as random oversampling and undersampling) when an imbalanced dataset is used for training a classifier. Consequently, cost-sensitive learning in DNNs [15, 18] has recently become more popular and a competitive option to the data resampling method when dealing with imbalanced data learning.

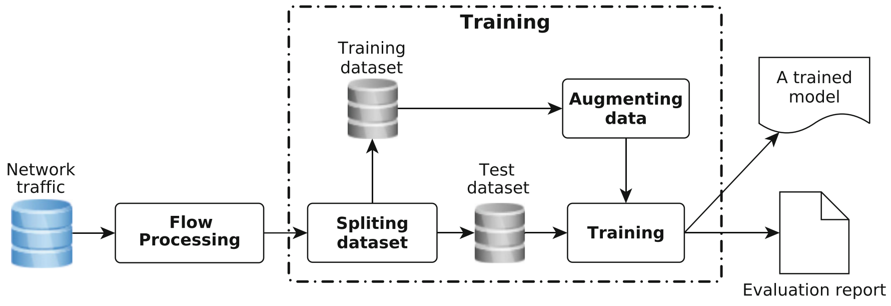


**Fig. 8.3** Training process

**Splitting dataset** We split the dataset in two subsets: one for training and one for evaluation. We prepare the training dataset by processing every packet in the flows using the procedure described in Sect. 8.2.1.3. We denote a flow dataset as $D = \{(F_T^{(j)}, y_j)\}$ for $1 \leq j \leq N$, where $N$ represents the total number of flows $F_T$ and their corresponding labels $y$.

**Augmenting data** Since our objective is to train the classifier capable of reliably detecting the attack flow after observing the first few packets out of a given flow, we extend the dataset by cumulatively creating short segments of a flow at different lengths.

The process of extending the training dataset by generating more data (e.g., network flows) from existing data is called *data augmentation* [34]. According to a predefined segmentation rate $s_r$, we create the shortest segment of a given flow containing only the first few packets of the flow; subsequently, we create more segments based on the flow by cumulatively adding more packets. Segmentation rate $s_r$ is a hyper-parameter such that $0 < s_r < 1$. It is used to calculate the segment size $s_z = \lceil s_r * T \rceil$ for a given flow, where $T$ is the length (i.e., total number of packets) of a flow. This parameter value controls the number of segments generated per flow, for instance, more segments per flow will be generated as the value of $s_r$ gets smaller. Suppose we have a flow $F_T^{(j)} = \{P_1, P_2, \ldots, P_T\}$, then the set of segments of this flow is as follows:

$$\{F_{t=k*s_z}^{(j)} | k = 1, 2, \ldots, \lfloor \frac{T-1}{s_z} \rfloor\}$$

where all the segments have the same label $y_j$ as the original flow does.

We would like to point out that setting the segment size $s_z$ to a fixed value for every flow in the training dataset would be suboptimal because we will end up with more segments for longer flows and fewer for shorter ones in the dataset. Since the length of flows can vary over a wide range in a dataset, this can worsen the class imbalance problem in the dataset. Therefore, in this work, we employ the segmentation rate $s_r$ that is used to calculate the segment size $s_z$ for a given flow based on its length $T$. This allows us to generate roughly the same number of segments of flows with different lengths. This method is adapted from [11] and can be used to generate segments based on different cumulative factors, such as 2 or 3.

For example, consider three flows with different lengths: $F_6^{(1)}$, $F_{15}^{(2)}$, and $F_{70}^{(3)}$. We set the segmentation rate $s_r$ to 0.25. The segment sizes $s_z$ for $F_6^{(1)}$, $F_{15}^{(2)}$, and $F_{70}^{(3)}$ are 2, 4, and 18, respectively. Table 8.1 lists the segments of the flows generated by the data augmentation process.

Data augmentation is only applied to the flows in the training dataset. The dataset is extended by including the generated flow segments.

**Training** We train our early classifier to learn a mapping function $\mathcal{H} : F_t^{(j)} \rightarrow y_j$, where $t \leq T$. In other words, the classifier should be able to predict the class label

**Table 8.1** Flow segments with respect to the segmentation rate $s_r = 0.25$

| No. | Flows | Flow segments |
|---|---|---|
| 1 | $F_6^{(1)} = \{P_1, P_2, \ldots, P_6\}$ | $\{P_1, P_2\}$ |
| 2 | | $\{P_1, P_2, P_3, P_4\}$ |
| 3 | $F_{15}^{(2)} = \{P_1, P_2, \ldots, P_{15}\}$ | $\{P_1, P_2, P_3, P_4\}$ |
| 4 | | $\{P_1, P_2, \ldots, P_8\}$ |
| 5 | | $\{P_1, P_2, \ldots, P_{12}\}$ |
| 6 | $F_{70}^{(3)} = \{P_1, P_2, \ldots, P_{70}\}$ | $\{P_1, P_2, \ldots, P_{18}\}$ |
| 7 | | $\{P_1, P_2, \ldots, P_{36}\}$ |
| 8 | | $\{P_1, P_2, \ldots, P_{54}\}$ |

$y_j$ of a given flow $F_t^{(j)}$ by examining only the first $t$ packets. We have used the categorical cross-entropy loss function and Adam [19] optimizer for training our classifier. The model/classifier resulting from training is stored in the Model library, accompanied by a description of that model architecture, types of attacks supported, and accuracy.

The EARLY tool can use two different types of neural networks to detect network attacks, as discussed in the following:

*Convolutional neural networks* (1D CNNs) [20] are used to extract a good internal representation of network flows and provide it as an input to a fully connected or dense layer. We use a *softmax* layer [13] as the final layer of our network to calculate a probability distribution for target classes. The CNNs are used to extract relevant features from grid-shaped input data such as images and sequences. They are capable of modeling the spatial and temporal dependencies in the data by learning relevant convolution filters (i.e., a set of grid-shaped weights or trainable parameters). A convolution layer is composed of several convolution filters, and each filter is used to extract a certain feature from the input data. Thus, the output of a convolution layer is called a *feature map*.

The input data for 1D CNNs has two dimensions: The first dimension specifies the sequence of events (i.e., packets in a network flow), whereas the second dimension correlates to the individual features of an event (i.e., bytes of a packet). We have used *rectified linear unit* (ReLU) [30] as a nonlinear activation function for every neuron in the convolutional layer. Typically, each convolutional layer is followed by a pooling layer [13] to achieve translation invariance of the output returned by the convolutional layer. This layer reduces the temporal size of the output by replacing each fixed-size partition of it with a summary statistic (e.g., maximum or average) of the adjacent elements. The CNNs have a smaller number of trainable parameters than other types of artificial neural networks such as fully connected networks [20]. Therefore, they are less likely to overfit the training data than the fully connected networks which result in a better generalization.

After convolution and pooling operations, a given variable-length network flow is represented by a variable-length series of feature maps. We use a *global pooling layer* [23] to transform the series into a fixed-length vector, which is then provided as an input to fully connected layers to get the feature vector. Lastly, we apply a *softmax* layer to the feature vector in order to obtain a probability distribution for each class. Based on the probability distribution and the classification threshold, we make the final predictions. We denote this neural network architecture by $EARLY_{CNN}$.

*Recurrent neural networks* (RNNs) [13] are used for processing time series data such as a network flow $F_T = \{P_1, P_2, \dots, P_T\}$. In RNNs, each neural network node has a memory unit (i.e., also known as the hidden state) that represents the previous state of the network. The current hidden state $h_t$ is a function of the previous hidden state $h_{t-1}$ and the current input (i.e., a packet in our case) $P_t$: $h_t = f(h_{t-1}, P_t)$, where $t$ represents the current time step. Subsequently, the current hidden state $h_t$ is used to compute the final output of the node. In summary, an RNN layer uses the information learned from the previous time steps and the current input to produce the output.

In this work, we use gated recurrent unit (GRU), a variant of RNN, which performs better than classic RNN and LSTM (i.e., another RNN variant). Further, it can learn long-term sequential dependencies efficiently because it does not suffer from the vanishing gradient issue [8]. We replace the convolution layer with a GRU layer to extract the relevant features from the raw network traffic in the $EARLY_{CNN}$ architecture while keeping the rest of the architecture unchanged. We denote the new neural network architecture by $EARLY_{RNN}$.

### *8.2.3 Monitoring*

The *packet sniffer* component in our approach is responsible for network traffic monitoring in real time. It captures and forwards every inbound and outbound network packet to the flow processing component, as shown in Fig. 8.4. This component is implemented using the *libpcap*[1] library that provides a programming interface to capture packets passing through the network interfaces.

Ideally, we should capture and process every network packet for inspection to detect attack attempts. But such an approach would be resource-intensive and possibly impractical, particularly when dealing with high-speed network traffic. In order to sustain a high packet rate, in our approach, we capture and process only those network packets which are related to the type of attacks we would like to detect using the *libpcap* filters. For example, we can configure the library filters to capture only those packets whose destination port is 80. These filters, which are usually supported by the operating system kernel, improve performance by reducing the packet processing overhead.

A list of active flows is maintained along with the predictions corresponding to those flows made by our early flow classifier (see Fig. 8.5). Whenever a network flow is updated with a new packet, the early flow classifier makes a prediction (i.e., a probability distribution for output classes) using the neural network model. The

[1] https://github.com/the-tcpdump-group/libpcap

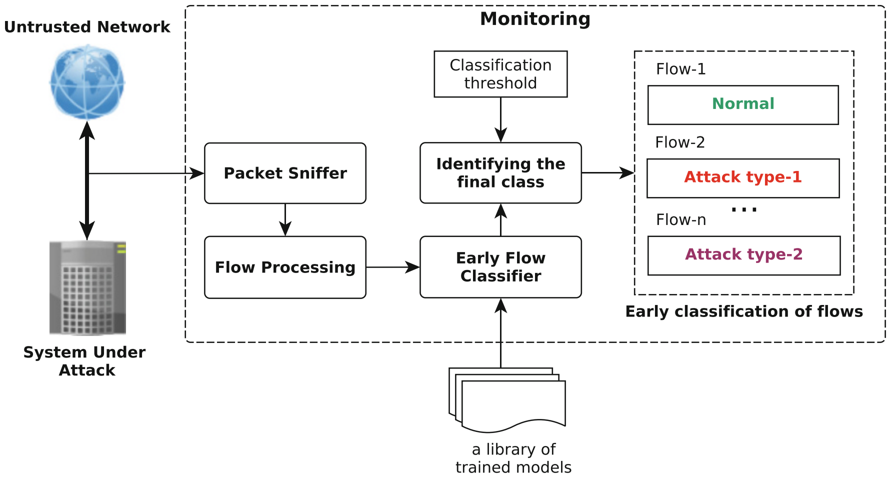


**Fig. 8.4** Monitoring

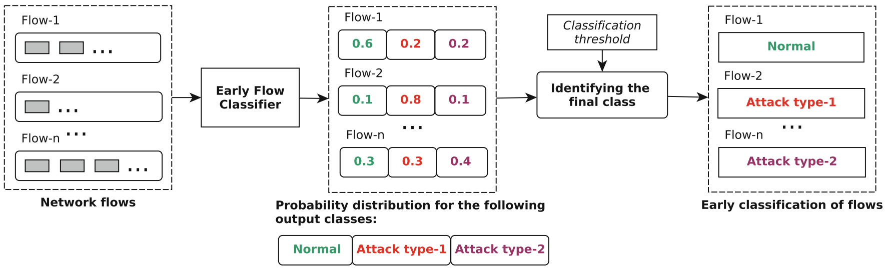
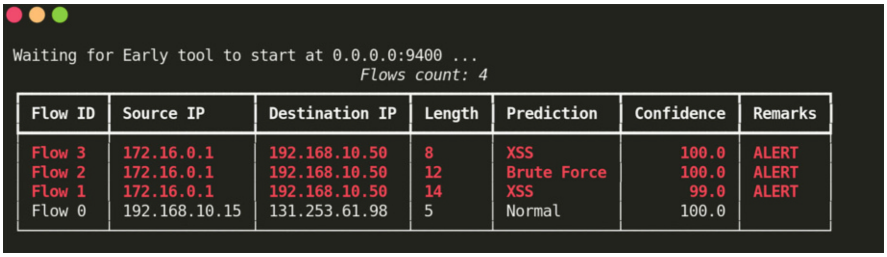


**Fig. 8.5** Early classification of flows

final class of a flow is a class that has a higher probability than other classes and the *classification threshold* $\in [0, 1)$. If none of the class probabilities is higher than a given threshold, our approach will return *Unknown* as the final class. As we increase the classification threshold, the number of false positives (i.e., a result that indicates a given flow is an attack when it is not) decreases, which improves the classification accuracy but degrades the earliness of the approach. The threshold is provided by a person such as a network administrator who observes the network traffic and is responsible for taking countermeasures against the attacks based on the classification results.

*Alerts* can be defined to notify external systems to automatically deploy countermeasures when the attack detection probability (or confidence) goes over a predefined threshold:

$$detection\ probability \geq threshold \rightarrow alert$$



```
Waiting for Early tool to start at 0.0.0.0:9400 ...
                                      Flows count: 4

| Flow ID | Source IP     | Destination IP | Length | Prediction  | Confidence | Remarks |
|---------|---------------|----------------|--------|-------------|------------|---------|
| Flow 3  | 172.16.0.1    | 192.168.10.50  | 8      | XSS         |      100.0 | ALERT   |
| Flow 2  | 172.16.0.1    | 192.168.10.50  | 12     | Brute Force |      100.0 | ALERT   |
| Flow 1  | 172.16.0.1    | 192.168.10.50  | 14     | XSS         |       99.0 | ALERT   |
| Flow 0  | 192.168.10.15 | 131.253.61.98  | 5      | Normal      |      100.0 |         |
```

**Fig. 8.6** User interface of the EARLY monitor

The EARLY monitor is available as open source at [2]. At the moment, pretrained models for Web attacks and MQTT attacks are available. A screenshot of the EARLY monitor interface is shown in Fig. 8.6, in which three out of four network flows are classified as malicious since the probability of prediction was higher than the specified threshold.

## 8.3 Evaluation

In this section, we evaluate how the two types of neural network architectures affect the performance of our approach by answering the following research questions:

- RQ1. How do the different neural network architectures affect the classification performance of our approach in terms of classifying the complete flows (i.e., flows with all the packets)?
- RQ2. Which neural network architecture is more effective in identifying the class of a given flow in real time by inspecting only the first few packets of the flow?

RQ1 investigates the classification performance, whereas RQ2 evaluates the performance of our approach when deployed in a real-time environment with respect to the neural network architectures. For evaluation, we use two datasets, one for web-based attacks (CICIDS2017) and one for MQTT-based attacks (MQTT-IDS-2020). We evaluate the performance of our approach using different neural network architectures against each dataset.

We use scikit-learn [31] library for data preprocessing and TensorFlow [1] library to train the neural network models.

### *8.3.1 Datasets*

#### 8.3.1.1 Web-Based Network Attack Detection

We use CICIDS2017 [33] dataset to evaluate the effectiveness of our approach. The dataset is composed of normal and seven types of attack flow (e.g., Heartbleed, botnet, web) along with the network packets corresponding to the flows. We use a specific part of the dataset that was captured on Thursday, July 6, 2017, and contains 29,309 network flows and 4,074,194 packets related to the following web attacks: (1) *SQL injection,* an attacker provides a string of SQL commands to be injected into the database; (2) *cross-site scripting (XSS),* an attacker injects a script into the web application code; and (3) *brute force,* an attacker tries a list of passwords to find the administrator's password. Table 8.2 lists the number of flows, the average flow length (i.e., number of packets), and the standard deviation (SD) of flow length per attack type in the dataset.

As one can notice, there is a large imbalance in the distribution of the flow classes; for example, there are roughly 1291.86 times more flows belonging to the normal class than the SQL injection class. We have used stratified sampling [10] to split the original dataset into training and test datasets using the 0.7:0.3 ratio (see Table 8.3). Unlike random sampling, stratified sampling creates the splits by maintaining the same percentage for each class as in the complete dataset.

We have augmented the training dataset using the segmentation rate $s_r = 0.1$. Table 8.4 lists the number of flows we obtained by applying the data augmentation technique described in Sect. 8.2.2.

We have observed that the header and payload length of 99% of packets in the dataset are less or equal to 40 and 356 bytes, respectively. To handle the packets with different header and payload lengths, we crop or pad them with zeros at the end to 48 and 400 bytes, respectively, as per the transformation step in Sect. 8.2.1.3. We scale all the packet bytes between 0 and 1 by dividing them by 255. In practice, scaling the input data helps machine learning algorithms converge faster [35].

**Table 8.2** Web-based flow dataset

| Class | No. of flows | Average flow length | SD of flow length |
|---|---|---|---|
| Normal | 27,129 | 124.39 | 6,508.44 |
| Brute force | 1, 507 | 18.43 | 64.20 |
| XSS | 652 | 11.48 | 46.75 |
| SQL injection | 21 | 5.71 | 3.25 |

**Table 8.3** Training and test datasets

| Class | Training | Test |
|---|---|---|
| Normal | 18,990 | 8,139 |
| Brute force | 1,055 | 452 |
| XSS | 456 | 196 |
| SQL injection | 15 | 6 |

**Table 8.4** Number of flows after augmenting the training dataset with $s_r = 0.1$

| Class | Original | Augmented |
|---|---|---|
| Normal | 18,990 | 92,468 |
| Brute force | 1,055 | 4,625 |
| XSS | 456 | 1,866 |
| SQL injection | 15 | 82 |

#### 8.3.1.2 MQTT-Based Attack Detection

EARLY was also evaluated in the context of MQTT using the MQTT-IDS-2020 [16] dataset in which the model was trained with both normal traffic and the following types of attacks: aggressive scan, user datagram protocol (UDP) scan, Sparta SSH brute force, and brute force, using a part of the dataset. Table 8.5 lists the number of flows, the average flow length (i.e., number of packets), and the SD of flow length per attack type in the dataset.

The total number of flows and packets related to those flows in the dataset is 3,397,121 and 32,177,882, respectively. Due to the limited amount of computing resources and time, it was not feasible for us to use the entire dataset for training and evaluation. Therefore, to speed up the training and evaluation processes, we uniformly sample 20,000 flows from each class. After random sampling, we split the dataset into two subsets using the ratio 0.7:0.3: training and test set. We have augmented the training dataset using the segmentation rate $s_r = 0.1$.

The header and payload length of 99% of packets in the dataset is less or equal to 40 and 34 bytes, respectively. To handle the packets with different header and payload lengths, we crop or pad them with zeros at the end to 40 and 50 bytes, respectively, as per the transformation step in Sect. 8.2.1.3. We scale all the packet bytes between 0 and 1 by dividing them by 255.

## *8.3.2 Model Architectures*

In this section, we discuss the neural network architectures used for detecting network attacks. We set the batch size to 32, which is the number of flows included

**Table 8.5** MQTT-based flow dataset

| Class | No. of flows | Average flow length | SD of flow length |
|---|---|---|---|
| Normal | 363,495 | 5.81 | 25.57 |
| MQTT brute force | 2,000,211 | 4.99 | 3.00 |
| Aggressive scan | 20,025 | 2.03 | 0.89 |
| UDP scan | 10 | 1.10 | 0.31 |
| Sparta SSH brute force | 1,013,380 | 19.45 | 4.41 |

in a mini-batch during neural network training. The total number of training epochs is set to 50.

#### 8.3.2.1 $EARLY_{CNN}$

This neural network model is made of a convolution block. The block contains the following layers in the specified order: 1D CNN layer with kernel size 1, valid padding, *ReLU* activation, and bias, layer normalization, and average pooling layer of size 2 with the same padding. We perform global average pooling to flatten the series of feature maps to a fixed-length vector, which is then provided as input to a fully connected layer to get the feature vector. Finally, we use a softmax layer to obtain a probability distribution for each class. We make the final predictions based on the probability distribution and the classification threshold.

Figure 8.7 portrays the architectures of the $EARLY_{CNN}$ model for the CICIDS2017 and MQTT-IDS-2020 datasets. The total number of trainable parameters of the models for the CICIDS2017 and MQTT-IDS-2020 datasets is 16,804 and 14,917, respectively. The label on the arrow from the *Input* to *Conv1D* layer in the figure specifies the dimensions of the input provided to the model. The input has three dimensions: the number of flows in a mini-batch, the number of packets in a flow, and the number of bytes or features representing a packet in a flow. Since we do not fix the number of packets required in a flow to make a prediction, the second dimension in the label (e.g., `32x?x448` in Fig. 8.7a) is left open.

#### 8.3.2.2 $EARLY_{RNN}$

This neural network model has a GRU layer with *Tanh* activation. The return sequence is set to true for the layer, which means that the layer will return the hidden state output for each time step (or packet). We perform global average pooling to flatten the series of hidden states to a fixed-length vector. Finally, we use a softmax layer to obtain a probability distribution for each class. Based on the probability distribution and the classification threshold, we make the final predictions.

Figure 8.8 portrays the architectures of the $EARLY_{RNN}$ model for the CICIDS2017 and MQTT-IDS-2020 datasets. The total number of trainable parameters of the models for CICIDS2017 and MQTT-IDS-2020 is 46,404 and 12,069, respectively.

### *8.3.3 Evaluation Metrics*

Evaluating the classification performance of a machine learning-based approach on an imbalanced dataset is a challenging task [45]. The majority of the existing IDSs

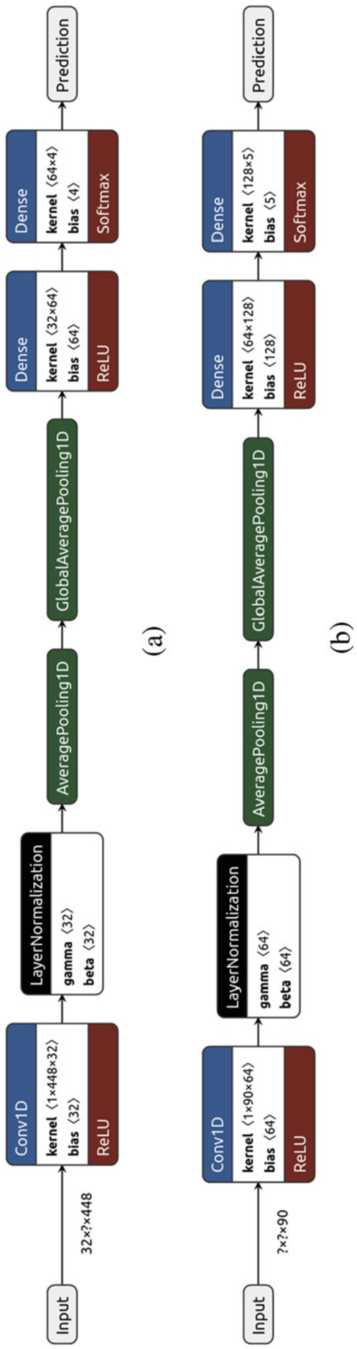


**Fig. 8.7** $EARLY_{CNN}$ neural network architectures. (**a**) Neural network architecture for CICIDS2017. (**b**) Neural network architecture for MQTT-IDS-2020

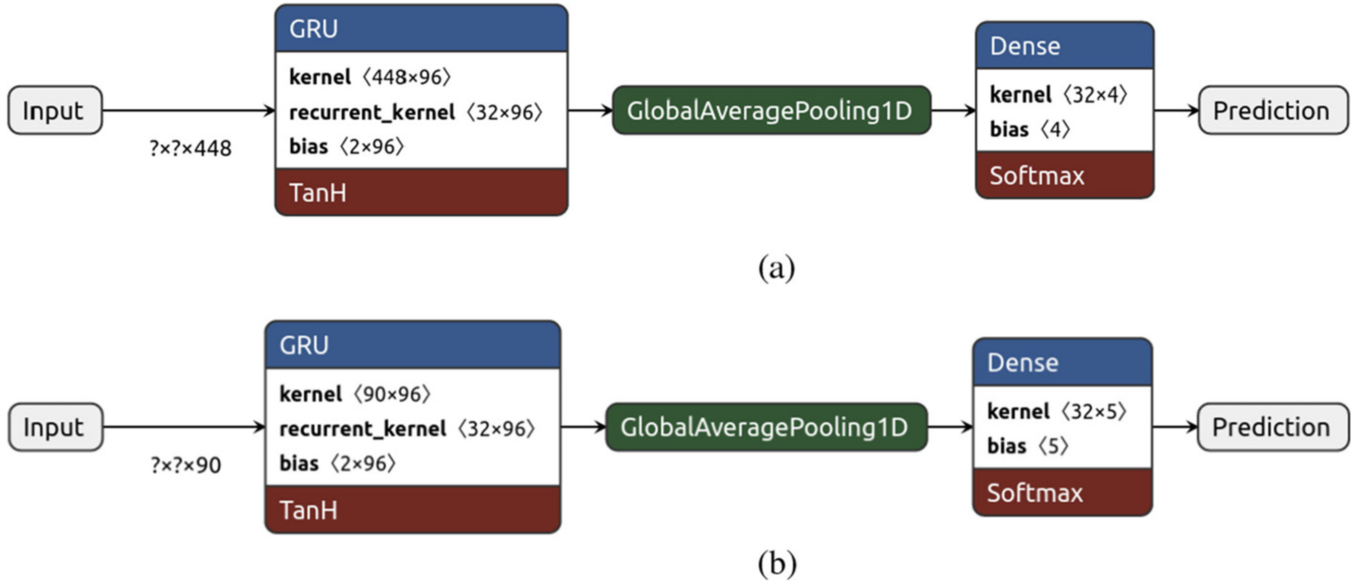


**Fig. 8.8** $EARLY_{RNN}$ neural network architectures. (**a**) Neural network architecture for CICIDS2017. (**b**) Neural network architecture for MQTT-IDS-2020

using machine learning have reported the performance of their approaches using traditional metrics such as accuracy and F1-score [4]. These metrics are designed to evaluate the performance of a classifier on balanced datasets. They do not work well when there is a large imbalance in the distribution of the classes in the dataset [45]. Therefore, we evaluate our tool, in the context of a web-based application and an MQTT Broker application, using the following metrics:

- *Precision* calculates the percentage of instances identified as positive that were correctly classified.
- *Recall* (i.e., also known as detection rate) computes the percentage of actual positive instances that were correctly classified.
- *False positive rate* (FPR) (i.e., also known as false alarm rate) estimates the proportion of negative observations wrongly predicted as positive over the total number of negative observations.
- *Balanced accuracy* (BA) is the arithmetic mean of recall obtained on each class.
- *Bookmaker informedness* (BM) is defined as the probability that the classifier will make a correct decision as opposed to random guessing.
- *Prediction time* indicates the average time needed by the tool to detect a security attack.
- *Earliness* specifies how early the correct class of a flow can be predicted ahead of the end of the flow. We define the *earliness* as:

$$Earliness = \begin{cases} \dfrac{T-t}{T-1} & \text{if } T > 1 \\ 1 & \text{if } T = 1 \end{cases} \tag{8.2}$$

  where $t$ is the minimum number of packets required to correctly predict the class of a given flow and $T$ is the total number of packets in the flow. Since this metric aims to evaluate the earliness of the prediction instead of its quality, this metric is only applied to those flows that are correctly classified and $t \leq T$.

All the metrics mentioned above can have values between 0 and 1. Higher values of precision, recall, BA, and BM and lower values of FPR indicate better classification performance of a classifier. The earliness value lies between 0 and 1, with extreme values 1 and 0 reached in case a classifier can accurately classify a given flow by analyzing only the first packet and all the packets of the flow, respectively.

### *8.3.4 RQ1: Classification Performance*

In order to answer this research question, we have trained and evaluated the architectures against the independent test set discussed in Sect. 8.3.1. We used tenfold cross-validation on the training dataset to fine-tune the hyper-parameter values and model selection for both neural network architectures. For statistical reasons, the evaluation procedure is repeated 30 times, and every time, we randomly shuffle the datasets to remove any ordering bias before splitting it into training and test set. We set the *classification threshold* to 0, which means that the final class of a flow is a class that has a higher probability than other classes.

A perfect IDS has a 1.0 recall at 0.0 FPR for every class, which means that it can identify all flows correctly without any miss-detection. Nevertheless, in reality, such flawless IDSs are empirically not feasible or very difficult to attain in a real-time environment because of the complexity and large volume of network traffic.

Table 8.6 lists the training time of the neural networks per dataset. One can notice that the training times of the $EARLY_{RNN}$ model are higher than the $EARLY_{CNN}$ for both datasets. Table 8.7 shows the achieved performance of our tool on both datasets using the $EARLY_{CNN}$ architecture. For the CICIDS2017 dataset, our tool gives the highest detection rate or recall of 0.911 at an FPR of 0.008 for the *XSS* attack type among all the other attack types. In other words, our tool correctly identifies 91.1% of the *XSS* attack flows in the test dataset and wrongly identifies less than 1% of other types of flows as *XSS* attack flows.

The tool has performed well also for the other types of attacks, even though the number of training flows for the attack types is low. For example, the number of training flows for *Brute force* and *XSS* attack types is only 5.1% and 2.2% of the total number of original training flows. One can notice that the approach has performed poorly for the *SQL injection* attack type. The main reason is that the number of samples of the attack type is significantly small (i.e., 0.07% of the total number of training samples). Thus, the model has a limited capacity to learn the attack type.

For the MQTT-IDS-2020 dataset, our tool gives the highest detection rate of 0.997 at an FPR of 0.008 for the *MQTT brute force* attack type among all the other attack types. As expected, the tool did not perform well for the *UDP scan* attack type because the number of training samples of the attack type was just 7.

Table 8.8 shows the achieved performance of our tool on both datasets using the $EARLY_{RNN}$ architecture. For the CICIDS2017 dataset, our tool gives the highest detection rate of 0.916 at an FPR of 0.003 for the *Brute force* attack type among all other attack types. For the MQTT-IDS-2020 dataset, our tool gives the highest detection rate of 0.999 at a FPR of 0.002 for the *MQTT brute-force* attack type among all the other attack types. The tool has performed poorly for the *SQL injection* and *UDP scan* attack types for the CICIDS2017 and MQTT-IDS-2020 datasets, respectively, due to the inadequate number of training samples. Table 8.9 lists the balanced accuracy scores attained by both neural network architectures. For both datasets, $EARLY_{RNN}$ has achieved higher balanced accuracy scores

**Table 8.6** Training times

| Dataset | Architecture | Training time (mins) |
|---|---|---|
| CICIDS2017 | $EARLY_{CNN}$ | 17.06 |
| | $EARLY_{RNN}$ | 29.96 |
| MQTT-IDS-2020 | $EARLY_{CNN}$ | 40.93 |
| | $EARLY_{RNN}$ | 54.05 |

**Table 8.7** Classification performance of $EARLY_{CNN}$

| Dataset | Class | Precision | Recall | FPR | BM |
|---|---|---|---|---|---|
| CICIDS2017 | Normal | 0.996 | 0.944 | 0.054 | 0.891 |
| | Brute force | 0.720 | 0.828 | 0.051 | 0.778 |
| | XSS | 0.754 | 0.911 | 0.008 | 0.904 |
| | SQL injection | 0.343 | 0.528 | 0.003 | 0.525 |
| MQTT-IDS-2020 | Normal | 0.707 | 0.584 | 0.095 | 0.488 |
| | MQTT brute force | 0.979 | 0.997 | 0.008 | 0.989 |
| | Aggressive scan | 0.812 | 0.815 | 0.055 | 0.760 |
| | UDP scan | 0.004 | 0.422 | 0.038 | 0.384 |
| | Sparta SSH brute force | 0.809 | 0.778 | 0.066 | 0.712 |

**Table 8.8** Classification performance of $EARLY_{RNN}$

| Dataset | Class | Precision | Recall | FPR | BM |
|---|---|---|---|---|---|
| CICIDS2017 | Normal | 0.996 | 0.995 | 0.052 | 0.944 |
| | Brute force | 0.905 | 0.916 | 0.003 | 0.913 |
| | XSS | 0.823 | 0.916 | 0.004 | 0.912 |
| | SQL injection | 0.403 | 0.733 | 0.001 | 0.732 |
| MQTT-IDS-2020 | Normal | 0.827 | 0.758 | 0.053 | 0.705 |
| | MQTT brute force | 0.995 | 0.999 | 0.002 | 0.997 |
| | Aggressive scan | 0.938 | 0.987 | 0.022 | 0.965 |
| | UDP scan | 0.092 | 0.211 | 0.000 | 0.211 |
| | Sparta SSH brute force | 0.833 | 0.853 | 0.058 | 0.795 |

**Table 8.9** Balanced accuracy of both neural network architectures

| Dataset | Architecture | Balanced accuracy |
|---|---|---|
| CICIDS2017 | $EARLY_{CNN}$ | 0.803 |
| | $EARLY_{RNN}$ | 0.890 |
| MQTTIDS | $EARLY_{CNN}$ | 0.719 |
| | $EARLY_{RNN}$ | 0.762 |

than $EARLY_{CNN}$. In conclusion, the answer to research question RQ1 is that the $EARLY_{RNN}$ architecture has performed better in terms of classification accuracy than $EARLY_{CNN}$ for both datasets.

### *8.3.5 RQ2: Earliness Performance*

This research question aims to study the performance of our tool in detecting attacks as early as possible in a real-time environment. In our opinion, a real-time IDS should satisfy the following two requirements: First, the IDS should be able to process the data (i.e., network packets) as fast as it is being produced under real-life circumstances. Second, the minimum number of packets (MNP) required to accurately predict the class of a given flow should be less than the total number of packets in the flow.

To answer this question, we ran two replay sessions per dataset where we reproduced the network traffic captured and not previously used for training in the dataset against the $EARLY_{CNN}$ and $EARLY_{RNN}$ architectures. During the replay session, we monitored packet inter-arrival times (IAT), processing times required by our tool to make predictions when using the different neural network architecture, and the MNP. Our tool and the software that replayed the traffic ran on different machines. Each machine featured an Intel Core i9-10900X CPU, 64 GB of memory, RTX 3090 graphics card, and Ubuntu 20.04 Operating System. The machines were connected via a 1Gb Ethernet connection in an isolated environment to reduce network latency.

Table 8.10 shows the duration of the replay sessions, the number of packets retransmitted, the average packet IAT, and the prediction time per packet for each neural network architecture. In the case of CICIDS17, on average, the tool with the $EARLY_{CNN}$ architecture was able to make a prediction in 0.04 milliseconds per packet, for example, if a flow has four packets, the tool would take 0.24 milliseconds to predict its class that is seven times faster than $EARLY_{RNN}$.

Tables 8.11 and 8.12 show the earliness, the MNP, and the average flow length per class for $EARLY_{CNN}$ and $EARLY_{RNN}$, respectively. The results show that our tool can detect the class of a given flow by inspecting roughly only one to three packets in most of the cases. Further, one can notice that $EARLY_{RNN}$ has

**Table 8.10** Replay sessions

| Dataset | Duration (sec) | Packets retransmitted | Packet IAT (ms) | Architecture | Prediction time (ms) |
|---|---|---|---|---|---|
| CICIDS2017 | 29,004 | 4,074,195 | 7.11 | $EARLY_{CNN}$ | 0.06 |
| | | | | $EARLY_{RNN}$ | 0.42 |
| MQTT-IDS-2020 | 16,614 | 32,144,887 | 0.51 | $EARLY_{CNN}$ | 4.18 |
| | | | | $EARLY_{RNN}$ | 4.30 |

**Table 8.11** Earliness metric and the average minimum number of packets required (MNP) to predict the flow class for $EARLY_{CNN}$

| Dataset | Class | Earliness | MNP | Average flow length |
|---|---|---|---|---|
| CICIDS2017 | Normal | 0.991 | 2.11 | 124.39 |
| | Brute force | 0.936 | 2.11 | 18.43 |
| | XSS | 0.917 | 1.86 | 11.48 |
| | SQL injection | 0.509 | 3.31 | 5.71 |
| MQTT-IDS-2020 | Normal | 0.708 | 2.40 | 5.81 |
| | MQTT brute force | 0.991 | 1.03 | 4.99 |
| | Aggressive scan | 0.848 | 1.15 | 2.03 |
| | UDP scan | 0.525 | 1.04 | 1.10 |
| | Sparta SSH brute force | 0.689 | 6.73 | 19.45 |

**Table 8.12** Earliness metric and the average minimum number of packets required (MNP) to predict the flow class for $EARLY_{RNN}$

| Dataset | Class | Earliness | MNP | Average flow length |
|---|---|---|---|---|
| CICIDS2017 | Normal | 0.994 | 1.74 | 124.39 |
| | Brute force | 0.931 | 2.20 | 18.43 |
| | XSS | 0.886 | 2.19 | 11.48 |
| | SQL injection | 0.712 | 2.31 | 5.71 |
| MQTT-IDS-2020 | Normal | 0.922 | 1.03 | 5.81 |
| | MQTT brute force | 0.999 | 1.00 | 4.99 |
| | Aggressive scan | 0.974 | 1.02 | 2.03 |
| | UDP scan | 0.467 | 1.05 | 1.10 |
| | Sparta SSH brute force | 0.778 | 5.09 | 19.45 |

higher earliness than $EARLY_{CNN}$ and it requires fewer packets than $EARLY_{CNN}$ to correctly predict the class of a given flow.

In summary, $EARLY_{RNN}$ has outperformed $EARLY_{CNN}$ in terms of classification performance, earliness, and MNP; however, $EARLY_{CNN}$ takes less time to make to predict the class of a given flow.

## 8.4 Related Work

Recently, a number of deep learning-based IDS approaches have been proposed. Most of these approaches (e.g., [6, 14, 25, 26, 32, 37, 39]) rely on flow-based statistical features extracted by analyzing all the packets in a given flow such as total bytes, packet count, IP addresses, and port numbers. In contrast, the proposed approach aims to extract relevant features from raw network traffic data that can be used to reliably detect attack flows by analyzing the partial information already available of the flows during the early phase of attacks. In this section, we focus on some of the most important and recent related works on IDS that use machine learning to classify network attacks by extracting the relevant features from raw network traffic data.

Zhang et al. [43] proposed an IDS based on a convolutional neural network, named parallel cross-convolutional neural network (PCCN). They use the network traffic data to extract features, but they restrict the number of packets in a flow to 5. The authors mention that the PCCN network structure meets the real-time requirements of network IDS; however, they neither further discuss nor evaluate this aspect of the approach.

Zhu et al. [46] presented a hierarchical network IDS based on unsupervised clustering using deep auto-encoder and Gaussian mixture model. The proposed model comprises two sub-models: the first sub-model detects abnormal traffic in real time, and the second identifies the attack categories of abnormal traffic detected by the first one. They employ the feature processing method from PCCN approach [43] to obtain features for their IDS. The authors state that essential features are extracted based on the first few packets, which guarantee real-time network IDS. However, they neither discuss how the approach achieves real-time IDS nor evaluate this aspect of the approach. Further, they report the performance of their approach in terms of accuracy, F1-scores, and AUC averaged over all the classes, which could be misleading when the class distribution is imbalanced [45].

Zhang et al. [44] proposed a network IDS that integrates CNN and LSTM neural network structures to learn the spatial and temporal features of flows. Similar to our approach, they use network traffic data to extract features. However, they restrict the number of packets in a flow to 10, whereas we do not limit the number of packets in a flow and, in addition, analyze all the packets available in order to make an informed prediction. Further, they also report the performance of their approach with respect to the accuracy, F1-scores, and AUC averaged over all the classes, which could be misleading when the class distribution is imbalanced [45].

Yin et al. [41] proposed an IDS using RNNs. They evaluated the approach using the NSL-KDD dataset. They utilized 38 numeric and 3 non-numeric statistical features in the dataset. Similarly, Xu et al. [40] employed a GRU-based model to detect network attacks. The model is trained on statistical features and tested using KDD 99 and NSL-KDD datasets These datasets are considered outdated, and they lack raw network traffic data [4, 33]. Therefore, we utilize CICIDS2017 for training and evaluation of our tool. Further, our tool extracts the relevant features from raw network traffic data in an end-to-end manner instead of relying on manual or flow-based statistical features in order to detect network attacks as early as possible.

Alsyaibani et al. [5] built an IDS using a GRU-based model. They utilized the CICIDS 2017 dataset for training and evaluation of the model. All the labels in the dataset were converted into 0 and 1 to represent attacks and benign traffic. Their model is trained on the flow-based statistical features; on the other hand, we let our model extract the relevant features from raw network traffic data.

Zhang et al. [42] proposed a multiple-layer representation learning model for network IDS by combining CNN with gcForest. They propose a new data encoding scheme based on P-Zigzag to encode a network packet into a two-dimensional gray-scale image for classification. In contrast to our approach, this approach classifies packets instead of flows.

López-Vizcaíno et al. [24] defined the early intrusion detection problem by grouping network packets into data flows, where each flow is labeled as an attack or normal traffic depending on the intent of its packets. The ideas and concepts in this work are very relevant to our work. The authors propose a new time-aware metric, named ERDE, where accurate predictions are penalized if they are made after a certain measuring point $o$ that is defined manually. This metric was initially proposed to measure the early detection of depressed individuals based on their posts on a social network. In contrast, our nonparametric *earliness* metric is designed specifically for network flows. The metric value ranges from 0 to 1, with extreme values 0 and 1 reached if a classifier can accurately classify a given flow by analyzing only the first packet and all the flow packets, respectively. In comparison to ERDE, we consider our metric to be more informative, comparable, and intuitive.

For evaluation, the authors above divide every flow into ten chunks containing 10% of the packets for each flow. A set of classifiers (i.e., Random Forest, J48, JRip, and PART) analyzes each chunk of flows sequentially, and it can produce three outputs: attack, normal, or delay. The objective is to detect an attack using as few chunks as possible. They utilize the feature extraction method from [28] that extracts traffic statistics, such as source port, IP, and MAC addresses, from every new packet transmitted over a network channel. Although the authors define the early intrusion detection problem in terms of network flows, they do not explain how they utilize the features extracted using a method (that does not consider network flows and processes each packet independently) in order to predict the class of a given flow. In contrast, we describe packet preprocessing steps and the features used for early classification in Sects. 8.2.1 and 8.2.2, respectively, in detail. The authors conclude that machine learning models do not perform well when they are used for early intrusion detection; however, our results show that our approach can identify attacks with a high degree of accuracy by analyzing the first few packets of a given flow.

In summary, to the best of our knowledge, the existing IDSs can detect a certain attack by inspecting the complete information related to the attack. This means that a system would only be able to detect an attack after it has been executed on the system under target and might have caused damage to the system. In contrast, our end-to-end early IDS can reliably detect attacks by analyzing the partial information already available in the early phase of attacks.

## 8.5 Conclusion

In this paper, we have presented an end-to-end early IDS that can predict and prevent network attacks in real time before they could cause any more damage to the system under attack.

The tool supports two types of classifier architectures, CNN-based and RNN-based. Regardless of the selected architecture, attack detection models are trained in a supervised manner to extract relevant features from raw network traffic data, instead of relying on a manual feature selection process used in most related approaches. We have evaluated the tool and its classifier architectures on two different datasets. For the evaluation, we have used a new metric, earliness, to quantify the earliness of the predictions made by the tool.

The results show that EARLY identifies attacks with a high degree of accuracy by analyzing roughly only one to three packets. Our approach has achieved overall 0.803 and 0.719, respectively, balanced accuracy. In terms of classification performance, earliness, and MNP, an RNN-based model outperformed a CNN-based model. However, the RNN-based model is approximately ten times slower than the CNN-based in predicting the class of a given flow. Further, the CNN-based model trains faster than the RNN-based one. In the future, we aim to evaluate our tool with other datasets containing encrypted traffic.

The main threat to external validity is that the evaluation might seem subjective because we have not compared our approach with other IDS approaches from the literature. However, as discussed in Sect. 8.4, we could not find any approach similar to ours that extracts the relevant features from raw network traffic data in an end-to-end manner instead of relying on manual or flow-based statistical features and detects network attacks as early as possible. Another threat to validity is that we have not evaluated the scalability of the tool to detect a large number of simultaneous attacks. This evaluation will be subject to future work.

**Acknowledgments** This work was made possible with funding from the European Union's Horizon 2020 research and innovation program, under Grant Agreement No. 957212 (VeriDevOps). The opinions expressed and arguments employed herein do not necessarily reflect the official views of the funding body.

## References

1. M. Abadi, A. Agarwal, P. Barham et al., *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems* (2015). https://www.tensorflow.org/, software available from tensorflow.org
2. T. Ahmad, D. Truscan, Early tool (2022). https://github.com/VeriDevOps/Earlytool
3. T. Ahmad, D. Truscan, J. Vain, I. Porres, Early detection of network attacks using deep learning, in *15th IEEE International Conference on Software Testing, Verification and Validation Workshops ICST Workshops 2022*, Valencia, Spain, 4–13 Apr 2022. IEEE (2022), pp. 30–39. https://doi.org/10.1109/ICSTW55395.2022.00020
4. Z. Ahmad, A. Shahid Khan, C. Wai Shiang, J. Abdullah, F. Ahmad, Network intrusion detection system: a systematic study of machine learning and deep learning approaches. Trans. Emerg. Telecommun. Technol. **32**(1), e4150 (2021). https://doi.org/10.1002/ett.4150
5. O.M.A. Alsyaibani, E. Utami, A.D. Hartanto, Intrusion detection system model based on gated recurrent unit to detect anomaly traffic (2021). https://doi.org/10.1109/ICOIACT53268.2021.9564003
6. G. Andresini, A. Appice, N.D. Mauro, C. Loglisci, D. Malerba, Multi-channel deep feature learning for intrusion detection. IEEE Access **8**, 53346–53359 (2020). https://doi.org/10.1109/ACCESS.2020.2980937
7. Y. Bengio, Deep learning of representations for unsupervised and transfer learning, in *Proceedings of ICML Workshop on Unsupervised and Transfer Learning. Proceedings of Machine Learning Research, PMLR*, Bellevue, Washington, USA, 02 Jul 2012, ed. by I. Guyon, G. Dror, V. Lemaire, G. Taylor, D. Silver, vol. 27, pp. 17–36. http://proceedings.mlr.press/v27/bengio12a.html
8. J. Chung, Ç. Gülçehre, K. Cho, Y. Bengio, Empirical evaluation of gated recurrent neural networks on sequence modeling. CoRR abs/1412.3555 (2014). http://arxiv.org/abs/1412.3555
9. B. Claise, B. Trammell, P. Aitken, Specification of the IP Flow Information Export (IPFIX) protocol for the exchange of flow information. RFC **7011**, 1–76 (2013)
10. W.G. Cochran, *Sampling Techniques*, 3rd edn. (John Wiley, 1977)
11. Z. Cui, W. Chen, Y. Chen, Multi-scale convolutional neural networks for time series classification. arXiv (2016)
12. P. Garcia-Teodoro, J.E.D. Verdejo, G. Maciá-Fernández, E. Vázquez, Anomaly-based network intrusion detection: techniques, systems and challenges. Comput. Secur. **28**(1–2), 18–28 (2009). https://doi.org/10.1016/j.cose.2008.08.003

13. I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning* (MIT Press, 2016). http://www.deeplearningbook.org
14. J. Gu, S. Lu, An effective intrusion detection approach using SVM with naïve bayes feature embedding. Comput. Secur. **103**, 102–158 (2021). https://doi.org/10.1016/j.cose.2020.102158
15. H. He, E.A. Garcia, Learning from imbalanced data **21**, 1263–1284 (2009). https://doi.org/10.1109/tkde.2008.239
16. H. Hindy, C. Tachtatzis, R. Atkinson, E. Bayne, X. Bellekens, MQTT-IOT-IDS2020: MQTT internet of things intrusion detection dataset (2020). https://doi.org/10.21227/bhxy-ep04
17. A.D. Khairkar, D.D. Kshirsagar, S. Kumar, Ontology for detection of web attacks, in *2013 International Conference on Communication Systems and Network Technologies*, pp. 612–615 (2013). https://doi.org/10.1109/CSNT.2013.131
18. S.H. Khan, M. Hayat, M. Bennamoun, F.A. Sohel, R. Togneri, Cost-sensitive learning of deep feature representations from imbalanced data. IEEE Trans. Neural Networks Learn. Syst. **29**(8), 3573–3587 (2018). https://doi.org/10.1109/TNNLS.2017.2732482
19. D.P. Kingma, J. Ba, Adam: A Method for Stochastic Optimization. arXiv e-prints arXiv:1412.6980 (2014)
20. Y. Lecun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition. Proc. IEEE **86**, 2278–2324 (1998). https://doi.org/10.1109/5.726791
21. J. Li, Y. Qu, F. Chao, H.P.H. Shum, E.S.L. Ho, L. Yang, *Machine Learning Algorithms for Network Intrusion Detection* (Springer International Publishing, Cham, 2019), pp. 151–179. https://doi.org/10.1007/978-3-319-98842-9_6
22. H. Liao, C.R. Lin, Y. Lin, K. Tung, Intrusion detection system: a comprehensive review. J. Netw. Comput. Appl. **36**(1), 16–24 (2013). https://doi.org/10.1016/j.jnca.2012.09.004
23. M. Lin, Q. Chen, S. Yan, Network in network. arXiv (2014)
24. M.F. López-Vizcaíno, F.J. Nóvoa, D. Fernández, V. Carneiro, F. Cacheda, Early intrusion detection for OS scan attacks, in *18th IEEE International Symposium on Network Computing and Applications, NCA 2019*, Cambridge, MA, USA, 26–28 Sept 2019, ed. by A. Gkoulalas-Divanis, M. Marchetti, D.R. Avresky (IEEE, 2019), pp. 1–5. https://doi.org/10.1109/NCA.2019.8935067
25. R.K. Malaiya, D. Kwon, S.C. Suh, H. Kim, I. Kim, J. Kim, An empirical evaluation of deep learning for network anomaly detection. IEEE Access **7**, 140806–140817 (2019). https://doi.org/10.1109/ACCESS.2019.2943249
26. N. Marir, H. Wang, G. Feng, B. Li, M. Jia, Distributed abnormal behavior detection approach based on deep belief network and ensemble SVM using spark. IEEE Access **6**, 59657–59671 (2018). https://doi.org/10.1109/ACCESS.2018.2875045
27. K. McCarthy, B. Zabar, G. Weiss, Does cost-sensitive learning beat sampling for classifying rare classes? (2005). https://doi.org/10.1145/1089827.1089836
28. Y. Mirsky, T. Doitshman, Y. Elovici, A. Shabtai, Kitsune: an ensemble of autoencoders for online network intrusion detection. CoRR abs/1802.09089 (2018). http://arxiv.org/abs/1802.09089
29. B. Mukherjee, L. Heberlein, K. Levitt, Network intrusion detection. IEEE Netw. **8**(3), 26–41 (1994). https://doi.org/10.1109/65.283931
30. V. Nair, G.E. Hinton, Rectified linear units improve restricted Boltzmann machines, in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 21–24 June 2010, Haifa, Israel, ed. by J. Fürnkranz, T. Joachims (Omnipress, 2010), pp. 807–814. https://icml.cc/Conferences/2010/papers/432.pdf
31. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
32. S. Rajagopal, P.P. Kundapur, K.S. Hareesha, Towards effective network intrusion detection: from concept to creation on azure cloud. IEEE Access **9**, 19723–19742 (2021). https://doi.org/10.1109/ACCESS.2021.3054688

33. I. Sharafaldin, A.H. Lashkari, A.A. Ghorbani, Toward generating a new intrusion detection dataset and intrusion traffic characterization, in *Proceedings of the 4th International Conference on Information Systems Security and Privacy, ICISSP 2018*, Funchal, Madeira – Portugal, 22–24 Jan 2018, ed. by P. Mori, S. Furnell, O. Camp (SciTePress, 2018), pp. 108–116. https://doi.org/10.5220/0006639801080116
34. C. Shorten, T.M. Khoshgoftaar, A survey on image data augmentation for deep learning. J. Big Data **6**, 60 (2019). https://doi.org/10.1186/s40537-019-0197-0
35. J. Sola, J. Sevilla, Importance of input data normalization for the application of neural networks to complex industrial problems. **44**, 1464–1468 (1997). https://doi.org/10.1109/23.589532
36. N. Thai-Nghe, Z. Gantner, L. Schmidt-Thieme, Cost-sensitive learning methods for imbalanced data (2010). https://doi.org/10.1109/ijcnn.2010.5596486
37. M.F. Umer, M. Sher, Y. Bi, Flow-based intrusion detection: techniques and challenges. Comput. Secur. **70**, 238–254 (2017). https://doi.org/10.1016/j.cose.2017.05.009
38. G. Vigna, W.K. Robertson, D. Balzarotti, Testing network-based intrusion detection signatures using mutant exploits, in *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004*, Washington, DC, USA, 25–29 Oct 2004 (ACM, 2004), pp. 21–30. https://doi.org/10.1145/1030083.1030088
39. R. Vinayakumar, M. Alazab, K.P. Soman, P. Poornachandran, A. Al-Nemrat, S. Venkatraman, Deep learning approach for intelligent intrusion detection system. IEEE Access **7**, 41525–41550 (2019). https://doi.org/10.1109/ACCESS.2019.2895334
40. C. Xu, J. Shen, X. Du, F. Zhang, An intrusion detection system using a deep neural network with gated recurrent units. IEEE Access **6**, 48697–48707 (2018). https://doi.org/10.1109/ACCESS.2018.2867564
41. C. Yin, Y. Zhu, J. Fei, X. He, A deep learning approach for intrusion detection using recurrent neural networks. IEEE Access **5**, 21954–21961 (2017). https://doi.org/10.1109/ACCESS.2017.2762418
42. X. Zhang, J. Chen, Y. Zhou, L. Han, J. Lin, A multiple-layer representation learning model for network-based attack detection. IEEE Access **7**, 91992–92008 (2019). https://doi.org/10.1109/ACCESS.2019.2927465
43. Y. Zhang, X. Chen, D. Guo, M. Song, Y. Teng, X. Wang, PCCN: parallel cross convolutional neural network for abnormal network traffic flows detection in multi-class imbalanced network traffic flows. IEEE Access **7**, 119904–119916 (2019). https://doi.org/10.1109/ACCESS.2019.2933165
44. Y. Zhang, X. Chen, L. Jin, X. Wang, D. Guo, Network intrusion detection: based on deep hierarchical network and original flow data. IEEE Access **7**, 37004–37016 (2019). https://doi.org/10.1109/ACCESS.2019.2905041
45. Q. Zhu, On the performance of Matthews correlation coefficient (MCC) for imbalanced dataset. Pattern Recognit. Lett. **136**, 71–80 (2020). https://doi.org/10.1016/j.patrec.2020.03.030
46. Y. Zhu, D. Han, X. Yin, A hierarchical network intrusion detection model based on unsupervised clustering, in *MEDES '21: Proceedings of the 13th International Conference on Management of Digital EcoSystems, Virtual Event*, Tunisia, 1–3 Nov 2021, ed. by R. Chbeir, Y. Manolopoulos, L. Bellatreche, D. Benslimane, M. Ivanovic, Z. Maamar (ACM, 2021), pp. 22–29. https://doi.org/10.1145/3444757.3485098

# Chapter 9
# A Stream-Based Approach to Intrusion Detection

**Sylvain Hallé**

**Abstract** Integrating security in the development and operation of information systems is the cornerstone of SecDevOps. From an operational perspective, one of the key activities for achieving such an integration is the detection of incidents (such as intrusions), especially in an automated manner. However, one of the stumbling blocks of an automated approach to intrusion detection is the management of the large volume of information typically produced by this type of solution. Existing works on the topic have concentrated on the reduction of volume by increasing the precision of the detection approach, thus lowering the rate of false alarms. However, another less explored possibility is to reduce the volume of evidence gathered for each alarm raised. This chapter explores the concept of intrusion detection from the angle of complex event processing. It provides a formalization of the notion of pattern matching in a sequence of events produced by an arbitrary system, by framing the task as a runtime monitoring problem. It then focuses on the topic of incident reporting and proposes a technique to automatically extract relevant elements of a stream that explain the occurrence of an intrusion. These relevant elements generally amount to a small fraction of all the data ingested for an alarm to be triggered and thus help reduce the volume of evidence that needs to be examined by manual means. The approach is experimentally evaluated on a proof-of-concept implementation of these principles.

**Keywords** Event stream processing · Explainability · Intrusion detection · Pattern matching

S. Hallé (✉)
Laboratoire d'informatique formelle, Université du Québec à Chicoutimi, Chicoutimi, QC, Canada
e-mail: shalle@acm.org

A. Sadovykh et al. (eds.), *CyberSecurity in a DevOps Environment*,
https://doi.org/10.1007/978-3-031-42212-6_9

## 9.1 Introduction

Information systems can produce logs of various kinds. For instance, workflow management systems, CRM systems, and ERP platforms produce event logs in some common format based on XML. Financial transaction systems also keep a log of their operations in some standardized and documented format, as is the case for web servers such as Apache and Microsoft IIS. Network monitors also receive streams of packets whose various headers and fields can be analyzed. It has long been recognized that these logs can be used as a valuable source of data for the real-time monitoring of a system and the timely detection of misbehavior, incorrect, or malicious activities one can loosely call *attacks*.

Prior work in this area mostly focuses on the efficient detection of specific patterns in a stream of data elements; when such a pattern is found, an alarm is triggered, and any defensive or corrective action is taken over by a possibly distinct part of the infrastructure. However, merely reporting that a given pattern match has occurred, without further details, is seldom sufficient: the appropriate corrective actions to be executed, if any, most probably depend on information based on the specific instance of the match. As a simple example, if a port scanning attack is detected on some machine in an organization, not much can be done to thwart this attack without knowledge of the address of that machine. Thus, one must not only accurately report matches of attack patterns but also extract additional information on each match.

Alas, we shall see in Sect. 9.2 that selecting what to extract when a pattern match is found has been the subject of much less work than simply detecting the pattern in the first place. In many systems, the information to extract must be provided manually along with the description of the pattern and is hence hard-coded for each specific pattern. In addition, attack patterns are often assumed to be represented as finite-state automata [22] or variants thereof.

In this chapter, we formally describe an intrusion detection framework based on the notion of event streams and stream processors. This framework presents the first advantage of being generic. In Sect. 9.3, attack patterns are expressed on abstract events and are defined by an arbitrary "black box" function, called a *monitor*, that is only assumed to return a Boolean verdict when a match is found on the underlying stream of events. This model encompasses existing definitions of patterns based on finite-state automata but can also be used on signatures expressed in other notations as well, such as temporal logic, or even completely custom user-defined functions.

In addition, our proposed framework introduces the concept of *progressing subsequence*, defined in Sect. 9.4. When a pattern match is detected over a stream of events, a progressing subsequence identifies a subset of this stream corresponding only to the events that are genuinely relevant to the match. Such a subsequence is based on a formal definition that is independent of the actual notation in which the pattern is expressed. Therefore, it can be computed automatically for any pattern that can be represented by a monitor.

Finally, the proposed framework also accounts for the fact that a pattern may be expressed as a composition of stream processing units called *processors*, where the output stream produced by a processor can be set as the input stream of the next one. Thus, complex patterns can be obtained by chaining sequences of elementary processors. In Sect. 9.5, the chapter describes a technique to keep track of the relationship between input events that are consumed and output events that are produced, which makes it possible to relate specific output events to a set of input events all the way up the chain of composed processors, thus providing a rudimentary form of lineage tracking.

Taken together, these notions make the building blocks of a generic intrusion detection mechanism where patterns can be expressed as compositions of monitors being fed with an arbitrary data source (a system log, a packet capture, etc.) and where the relevant pieces of information from this source can automatically be extracted when a pattern match is detected. In addition to be grounded in precise theoretical definitions, the resulting framework has also been implemented as a proof-of-concept system leveraging an existing event stream processing engine called BeepBeep [30], which is described and tested in Sect. 9.6.

## 9.2 Related Work

The field of attack and intrusion detection is extremely broad and has been the focus of a large number of scientific works spanning multiple decades. There already exist numerous recent surveys on the topic of intrusion detection [15, 19, 31, 34, 38, 40, 44, 52, 63], and it is not in the scope of this paper to repeat the synthesis that the reader can find in these references. In particular, Tidjon et al. propose a detailed taxonomy of existing approaches in the field [54], which classifies detection systems into three broad categories:

1. *Anomaly-based* approaches, which correspond to a host of recent works attempting to discover trend deviations, outliers, or otherwise “uncommon” events or actions. The techniques involved include machine learning [39, 49], neural networks [6, 51], clustering [46, 47, 50], decision trees [4, 13], and regression [60, 61].
2. *Multi-agent-based* approaches where cooperating computation units both observe and interact with the environment in order to detect and prevent attacks [21, 35–37].
3. *Knowledge-based* approaches where a system is given a priori information about the expected behavior of the system, either in the form of rules [41], cases [18], or ontologies [59].

This chapter focuses on the latter category and in particular on the detection of attacks, intrusions, or other security-related incidents based on the definition of behavioral patterns. In its broadest sense, the operation of an information system periodically produces observable data elements we shall call *events*. An event can be the insertion of a message into a log, the reception or transmission of a network packet observed by a probe, or the execution of a particular system process. An *incident* is defined as any undesirable circumstance whose occurrence can be deduced from the observation of events produced by its operation. The condition that determines whether the incident occurs or not, based on the observed events, is called a *pattern*.

Note that in this context the "pattern" is a very generic concept; it can represent an arbitrary (computable) condition, and is not tied to the necessity of expressing it in the notation or language of some existing detection tool. In particular, although a linear sequence of events to be observed (e.g., $A$ followed by $B$, followed by $C$, etc.) obviously counts as a possible pattern, it is far from being the only or the most complex type of condition one may wish to evaluate on a set of events produced by an information system. One must also keep in mind that a pattern can be positive or negative: it may either represent the behavior of an attack (raising an alarm when the pattern is detected) or a security rule that can be violated by an attack (raising an alarm when a deviation from the expected pattern is observed). In logical terms, this depends on whether the undesirable situation is expressed by a condition $\varphi$ or by its negation $\neg\varphi$.

From a network security standpoint, a number of detection approaches follow this generic definition. Industrial-grade network intrusion detection tools, such as Snort [1] and Zeek [2], provide languages for expressing rules which, when triggered, indicate the presence of an attack. However, lesser known to security practitioners is the existence of a series of works grounded in formal methods and software verification, in a field called runtime verification (RV) [9].

In RV, a special process called a *monitor* is given a formal specification of some desirable property that a trace should fulfill. The monitor is then fed events, either directly from the execution of some instrumented system or by reading a prerecorded file, and is responsible for providing a verdict as to whether the trace satisfies or violates the property. Although a few works have applied RV in the particular context of detecting attacks on software vulnerabilities [22, 32, 33, 45, 57], as a rule, the use of monitors for the detection of incidents has been overlooked.

A first notable contender in this area is Orchids [22, 45]. It extracts data and events from multiple distributed sources, such as system calls, firewall actions, and logs from various server processes. Detection rules on these events are represented as nondeterministic finite automata. Since the automata are not deterministic, several optimization strategies must be implemented in order for the system to discard any paths in the automaton that are subsumed by other paths, such that the shortest run in the automaton be detected.

Other approaches have been proposed which use temporal logic as their underlying formal basis. For instance, R2U2 specializes in the detection of incorrect behavior in unmanned aerial systems [42]. Implemented on an FPGA for better performance, it monitors the execution of various inputs, including a GPS unit installed in the system, other sensor readings, and communications from the ground control station. Among the patterns monitored by the system and which represent "unusual" situations, R2U2 can observe illegal commands, temporary variations in the GPS signal strength, repeated navigation commands, and commands sent in a context where they make little sense (such as resetting the system while the device is in midair). Other monitoring systems based on temporal logic include Monid [43], TeStID [5], and the attack signature description language (ASDL) [62].

Other notations take their roots in formal modeling languages such as abstract state machines [14], the B [3], and Z [53] notation. For example, algebraic state-transition diagrams [55] have been used to represent attack patterns over streams of events and illustrated in Fig. 9.1. Similar to UML statecharts, ASTDs allow the definition of state-transition diagrams, where edges linking two states can be completed with guards (conditions that must hold for the transition to take place) and side effects (such as modifications to values of internal variables associated with each diagram).

Closer to the contribution of this chapter is LOLA, a stream-based specification language [20]. A LOLA specification is a set of equations over typed stream variables. Figure 9.2 shows an example of such a specification, taken from the original paper and summarizing most of the language's features. It defines ten streams, based on three independent variables $t_1$, $t_2$, and $t_3$. A stream expression may involve the value of a previously defined stream. The values of the streams corresponding to $s_1$ to $s_6$ are obtained by evaluating their defining expressions place-wise at each position. The language provides the expression ite$(b; s_1; s_2)$, which represents an if-then-else construct: the value returned depends on whether the predicate of the first operand evaluates to true. It also allows a stream to be defined by referring to the value of an event in another stream $k$ positions behind, using the construct $s[-k, x]$. If $-k$ corresponds to an offset beyond the start of the trace, value $x$ is used instead. It was shown in earlier work that the formal model introduced in this chapter, based on computing units called *processors*, is more general than LOLA [26]; in other words, LOLA equations can be turned into equivalent pipelines made of the elementary processors presented in Sect. 9.3.1.

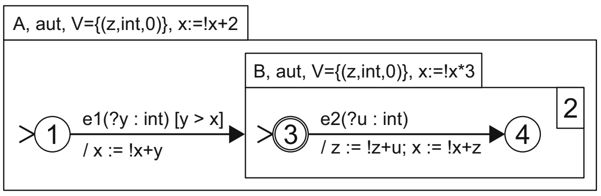


**Fig. 9.1** An example of an ASTD, from [55]

$$\begin{aligned} s_1 &= t_1 \vee (t_3 \leq 1) & s_4 &= \text{ite}(t_1; t_3 \leq s_4; \neg s_3) \\ s_2 &= ((t_3)^2 + 7) \mod 15 & s_5 &= t_1[+1; \text{false}] \\ s_3 &= \text{ite}(s_3; s_4; s_4 + 1) & s_6 &= s_9[-1; 0] + (t_3 \mod 2) \end{aligned}$$

**Fig. 9.2** An example of a LOLA specification showing various features of the language

All runtime monitoring approaches enumerated above have in common two features that make their direct use in attack detection problematic: First, a monitor can detect a single violation of a property in a sequence of events (typically the first occurrence of the violation). Consider, for instance, the simple property "an occurrence of $a$ must be immediately followed by $b$" on the input sequence $cacdaac$. A monitor will typically return a failing verdict ($\perp$) after ingesting the first three events, as, by this point, an $a$ not followed by a $b$ has been observed. However, one can see that there are actually two instances of this violation inside the trace, as the second occurrence of $a$ is also not followed by $b$. Yet, the monitor, after returning $\perp$ on some prefix of a sequence, returns $\perp$ forever. This makes it useless to detect any other subsequence of the input that could also be a violation of the pattern. As one can see, in a context of intrusion detection where a monitor is expected to be a nonterminating process, reporting a single violation is not desirable.

The second issue is that a monitor only reports a Boolean verdict. It reads a stream of events and eventually stops and emits the verdict "false" when a violation of the specification is observed. It does not identify element of the sequence that actually matters in the occurrence of the violation – all one can say is that they all lie in the interval from the beginning of the stream up to the point where the violation is reported and that potentially all of them may be relevant. Yet, we can see that not all events in the input are necessary to "explain" the violation; in the previous example, event $c$ at the start of the trace has no impact on the occurrence of any of the two violations; the same can be said of event $d$ occurring at position 4, which has no bearing on the occurrence of the second violation. This is not an issue if the goal is merely to detect a violation and act immediately upon it (e.g., by shutting the system down); however, if one is to report the violation and provide evidence of its occurrence (for *a posteriori* analysis or even for immediate handling by another layer of protection), the best a monitor can do is returning the *complete* sequence up to the point where the alarm was triggered, which is very likely to be unmanageable and of limited practical value.

## 9.3 Formalizing Intrusion Detection

To address the issues mentioned above, this chapter aims at providing the formal grounds for a generic and automated mechanism that can both express complex patterns or rules over sequences of events produced by an information system of some kind and calculate the subset of the whole input that is actually relevant for the occurrence of a match or a violation when it is detected.

We start in this section by introducing the formal definitions leading to the notion of *monitor*, which is an abstract entity able to recognize an arbitrary condition over a sequence of data elements called *events*. We then provide a first, basic algorithm for monitor-based pattern detection, which we shall extend and optimize in later sections.

### 9.3.1 Formal Preliminaries

Let $\Sigma$ be a set of abstract elements called *events*. We denote by $\Sigma^*$ the set of all finite sequences that can be created from elements of $\Sigma$. For two sequences $\overline{\sigma}, \overline{\sigma}' \in \Sigma^*$, we denote by $\overline{\sigma}[i]$ the $i$-th event of $\overline{\sigma}$ (with indices starting at 1) and by $\overline{\sigma} \preceq \overline{\sigma}'$ the fact that $\overline{\sigma}$ is a prefix of $\overline{\sigma}'$. The notation $\overline{\sigma}[i..j]$ will denote the sequence of successive events of $\overline{\sigma}$ between its $i$-th and $j$-th positions (inclusive). We shall abuse notation and define $\overline{\sigma}[i..]$ to denote the suffix of $\overline{\sigma}$ starting at position $i$ and likewise $\overline{\sigma}[..i]$ to denote the prefix of $\overline{\sigma}$ ending at position $i$ (inclusive).

Given a sequence of events $\overline{\sigma} \in \Sigma^*$ of length $n$, a sequence $\overline{\sigma}'$ is said to be a subsequence of $\overline{\sigma}$ if there exists an order-preserving injection between the events of $\overline{\sigma}'$ and those of $\overline{\sigma}$; we note this as $\overline{\sigma} \sqsubseteq \overline{\sigma}'$. In other words, a subsequence is obtained by deleting any number of events from another sequence, but not changing the ordering of the remaining events. Thus, *ade* and *bcde* are subsequences of *abcdef*, but *adb* and *cabc* are not. An $(i, j)$-*loop* in a sequence $\overline{\sigma}$ is a subsequence of successive events $\overline{\sigma}[i..j]$ such that $\overline{\sigma}[i] = \overline{\sigma}[j]$. The removal of loop $(i, j)$ in $\overline{\sigma}$ is the subsequence defined as $\overline{\sigma}[1..i] \cdot \overline{\sigma}[j+1..]$. For example, the sequence *abcdbe* has a (2, 5)-loop, and its removal produces the sequence *abe*; the loop *bcdb* has been contracted into the single event *b*.

If $\Sigma_1, \ldots, \Sigma_n$ are event alphabets, a stream vector $\boldsymbol{\sigma} = \langle \overline{\sigma}_1, \ldots, \overline{\sigma}_n \rangle$ is an element of $(\Sigma_1 \times \cdots \times \Sigma_m)^*$; note that this imposes that each stream within the vector is of the same length. The $n$-uple of events at identical indices in each stream is called a *front*. A stream vector $\langle \overline{\sigma}_1, \ldots, \overline{\sigma}_n \rangle$ is a prefix of another vector $\langle \overline{\sigma}'_1, \ldots, \overline{\sigma}'_n \rangle$ if each $\overline{\sigma}_i$ is a prefix of $\overline{\sigma}'_i$. Given a $n$-stream vector $\boldsymbol{\sigma} = \langle \overline{\sigma}_1, \ldots, \overline{\sigma}_n \rangle$ and an $n$-uple $(\sigma_1, \ldots, \sigma_n)$, the concatenation $\boldsymbol{\sigma} \cdot (\sigma_1, \ldots, \sigma_n)$ is the $n$-uple $\langle \overline{\sigma}_1 \cdot \sigma_1, \ldots, \overline{\sigma}_n \cdot \sigma_n \rangle$; this notion can then easily be extended to the concatenation of $n$-stream vectors. The length of a stream vector $\boldsymbol{\sigma}$, noted $||\boldsymbol{\sigma}||$, is defined as the number of events contained in any stream of the vector.

In the following, we are concerned with functions that take as input a stream vector and produces as output another stream vector. Such functions have been called *processors* by Bédard and Hallé [10]. Formally, a processor is a function $\pi : (\Sigma_1 \times \cdots \times \Sigma_m)^* \to (\Sigma'_1 \times \cdots \times \Sigma'_n)^*$, with the condition that $\boldsymbol{\sigma} \preceq \boldsymbol{\sigma}'$ implies $\pi(\boldsymbol{\sigma}) \preceq \pi(\boldsymbol{\sigma}')$. This condition makes it possible for a processor to operate in a streaming fashion: input event fronts can be ingested one at a time, and any number of output event fronts resulting from this input can be appended at the end of the current output vector. Note that as per this condition, event fronts that have already been output cannot be taken back; only new event fronts can be appended. The values of $m$ and $n$ are, respectively, called the input and output *arity* of the processor. When the input and output arity of a processor have no impact on a definition, we shall simplify notation and denote a processor as a 1:1 function $\pi : \Sigma^* \to \Sigma'^*$. We will denote by $\Pi$ the set of processors.

As an example, consider a processor $\pi$ given an input sequence $\overline{\sigma}$ and producing the corresponding output sequence $\pi(\overline{\sigma})$ as illustrated in Fig. 9.3. This notation indicates that $\pi(a) = A$, $\pi(ab) = AB$, $\pi(abc) = AB$, and so on. As one can

$$\begin{array}{rl} \overline{\sigma} = & a \;\; b \; c \; d \; e \qquad f \; g \; h \\ \pi(\overline{\sigma}) = & A \;\; B \qquad C \; D \; E \; F \end{array}$$

**Fig. 9.3** The relationship between input and output events for sequence $\overline{\sigma}$ and a processor $\pi$

see, in some cases, the processor does not append a new output event after ingesting an input event; hence, $\pi(ab) = \pi(abc) = \pi(abcd) = AB$. In some other cases, the processor may append several events to the output after ingesting a single input event: adding $e$ to the trace $abcd$ causes three new events ($CDE$) to be output at once. A processor that always outputs $k$ events (for some $k > 0$) for each ingested event is called *k-uniform*.

## 9.3.2 Monitors

Processors are very generic units of computation for event streams. One particular use of processors is to detect the presence of a specific sequential pattern in a stream vector; this corresponds to the definition of a *monitor* already discussed in Sect. 9.2. More precisely, a monitor can be seen as a particular type of processor that receives a stream of input events and produces an output made of events in the set $\mathbb{B}_3 = \{\top, ?, \bot\}$. We say that a sequence $\overline{\sigma} \in \Sigma^*$ is a *match* for $\pi$ if $\pi(\overline{\sigma}) = \top$; a match is said to be suffix-minimal if none of its prefixes is also a match. Conversely, $\overline{\sigma}$ is a *non-match* if $\top$ is replaced by $\bot$ in the condition above.

Matches and non-matches are expected to be definitive: if a particular pattern is found after reaching some position in an input stream, then this pattern is obviously still present after appending further events to this stream. However, the converse is also true: if a monitor declares a non-match after reading a prefix of an input stream, it actually indicates that this pattern will never be found, regardless of the events that may come after. In other words, the output of a monitor is expected to be *monotonic*; if $\pi(\overline{\sigma})$ ends with the symbol $\bot$ (resp. $\top$) for a sequence $\overline{\sigma}$, then $\pi(\overline{\sigma} \cdot \overline{\sigma}')$ ends with $\bot$ (resp. $\top$) for any $\overline{\sigma}' \in \Sigma^*$. Hence, a monitor that returns a conclusive verdict ($\top$ or $\bot$) keeps this verdict forever.

This characteristic explains the presence of a third possible verdict, represented by the symbol "?." A monitor that produces this symbol for some input stream $\overline{\sigma}$ means that the presence or absence of the pattern cannot be decided with the input events ingested so far. In such a case, $\overline{\sigma}$ is said to be a *potential* match. It follows that the output stream of a monitor is always of the form $?^m \cdot \top^n$ or $?^m \cdot \bot^n$ for $m, n \geq 0$. Since we are typically interested only in the latest event produced by the monitor, we define the verdict function $\nu_\pi : \Sigma^* \to \mathbb{B}_3$ as $\nu_\pi(\overline{\sigma}) \triangleq \pi(\overline{\sigma})[|\overline{\sigma}|]$.

As a simple example, consider the monitor $\pi_1 : \{a, b, c\}^* \rightarrow \mathbb{B}_3^*$ defined as follows:

$$\pi_1(\overline{\sigma}) \triangleq \begin{cases} ? & \text{if } |\overline{\sigma}| = 1 \text{ and } \overline{\sigma} = a \\ \bot & \text{if } |\overline{\sigma}| = 1 \text{ and } \overline{\sigma} \neq a \\ ? \cdot \top^{|\overline{\sigma}|-1} & \text{if } \overline{\sigma}[1..2] = ab \\ ? \cdot \bot^{|\overline{\sigma}|-1} & \text{otherwise} \end{cases}$$

One can observe that this monitor is defined to identify the pattern *ab* occurring at the start of a stream. Note how, for streams made of a single event, a non-match can already be declared if that symbol is not the expected *a*.

As a second example, consider a variation on the previous monitor, defined as follows:

$$\pi_2(\overline{\sigma} \cdot \sigma) \triangleq \begin{cases} \pi_2(\overline{\sigma}) \cdot \top & \text{if } \pi_2(\overline{\sigma})[|\overline{\sigma}|] = \top, \\ & \text{or } \overline{\sigma}[|\overline{\sigma}|] = a \text{ and } \sigma = b \\ \pi_2(\overline{\sigma}) \cdot ? & \text{otherwise} \end{cases}$$

Note how this time the monitor is defined recursively by specifying what output symbol to append to the existing output stream based on the event currently being ingested. One can observe that this monitor declares a match as soon as it sees event *a* immediately followed by *b* in the input, regardless of the position of that pattern. Since an input stream that does not contain the sequence *ab* can always contain it later if further events are appended, this monitor can declare matches but can never conclude to a non-match, as all stream prefixes that do not have *ab* are potential matches.

These two examples are very simple, as monitors are by far not restricted to finding straightforward patterns of consecutive events, and can be arbitrarily defined functions. As a last (and arguably convoluted) example, let us first define $f : \{a, b, c\} \rightarrow \mathbb{N}$ as $f(\overline{\sigma}) = |\overline{\sigma}|_a + 2|\overline{\sigma}|_b$, where the notation $|\overline{\sigma}|_x$ designates the number of events labeled $x$ in $\overline{\sigma}$. Then, consider the monitor defined as:

$$\pi_3(\overline{\sigma} \cdot \sigma) \triangleq \begin{cases} \pi_3(\overline{\sigma}) \cdot \top & \text{if } f(\overline{\sigma}') = 5 \text{ for some } \overline{\sigma}' \sqsubseteq \overline{\sigma} \cdot \sigma \\ \pi_3(\overline{\sigma}) \cdot ? & \text{if } f(\overline{\sigma} \cdot \sigma \cdot \overline{\sigma}') = 5 \text{ for some } \overline{\sigma}' \in \{a, b, c\}^* \\ \pi_3(\overline{\sigma}) \cdot \bot & \text{otherwise} \end{cases}$$

This monitor declares a match when one of its prefixes has the number of *a* and *b* in the stream following a specific arithmetic relation defined by function *f*. However, it must also declare a non-match as soon as no extension of the current input stream can ever make the input satisfy the condition. It turns out that $f(x) = 5$ is nothing but the linear Diophantine equation $x + 2y = 5$ [7], which admits only three positive integer solutions: $(1, 2)$, $(3, 1)$, and $(5, 0)$. Hence, $\pi(abb) = ??\top$, since the input stream at this point contains one $a$ and two $b$ and thus satisfies the condition. However, $\pi(aabb) = ???\bot$, as it is impossible to add any further symbols to this input that will make the number of $a$ and $b$ land on one of the three possible solutions.

This example is meant to illustrate the fact that the definition of a monitor gives complete leeway in how matches can be defined and may even require one to reason about the existence of possible extensions of the current input satisfying an arbitrary condition. Also note that monitors are considered as abstract monotonic functions and thus are not bound to any particular notation. A monitor can be defined using any of the formalisms already in use in the field of runtime verification, such as finite-state automata, regular expressions, and linear temporal logic, but can just as well be defined as functions like what was done in the examples above.

## *9.3.3 Pattern Detection as Monitoring*

Monitors can be used as tools to detect the presence of an arbitrarily complex pattern inside an input stream. However, as was argued earlier, they are limited to the detection of a single occurrence of the said pattern, which spans the interval from the start of the stream up to the first input event that triggers the production of the output $\top$. In a context of attack or intrusion detection, such a restriction is severely limiting. Intrusion detection is a long-running process, which must be performed continuously as a system operates and where multiple patterns are expected to be found throughout the lifetime of that system.

A single monitor cannot detect all such occurrences; however, one can evaluate this monitor separately on multiple subsequences of the input stream in order to find all matches. For a sequence $\overline{\sigma}$, the set of suffix-minimal matching subsequences of some monitor $\pi$ is defined as:

$$\mathcal{M}_\pi(\overline{\sigma}) \triangleq \{\overline{\sigma}[i..j] : 1 \leq i \leq j \leq |\overline{\sigma}| \text{ and } \overline{\sigma}[i..j] \text{ is a suffix-minimal match of } \pi\}$$

The set is made of all subsequences of successive events in $\overline{\sigma}$ that are suffix-minimal matches of $\pi$. By the definition of a (suffix-minimal) match, for a given $i \in [1, |\overline{\sigma}|]$, there exists at most one $j \geq i$ such that $\overline{\sigma}[i..j]$ is a minimal match for $\pi$; that is, any event of a given input stream is either not the start of a matching subsequence or the start of exactly one minimal match. It shall be noted, however, that matches can still overlap and need not be detected by a greedy procedure. Consider, for instance, the monitor $\pi$ that returns $\top$ on a sequence if it contains a $b$ that is immediately followed by a $c$. This monitor declares a match for the sequence $abaabca$; however, observe that the $b$ that is followed by a $c$ is the second one occurring in the sequence.

This construction induces a straightforward procedure to evaluate $\mathcal{M}_\pi(\overline{\sigma})$, detailed in Algorithm 1, and which takes as input a monitor $\pi$ and a stream of events $\overline{\sigma}$. The algorithm iterates over all ranges of successive events $[i, j]$ (lines 2–3), evaluates the monitor $\pi$ on that range and collects its last verdict (line 4), and adds the range $[i, j]$ to a set $\mathcal{M}$ if the verdict indicates a match (line 6). The break instruction at line 7 prevents suffixes of a match to be examined, thus limiting the output to suffix-minimal matches. As a matter of fact, one can easily observe that at line 11, $\mathcal{M} = \mathcal{M}_\pi(\overline{\sigma})$.

**Algorithm 1** Finding all suffix-minimal matches in $\overline{\sigma}$ for a monitor $\pi$

```
1: $\mathcal{M} \leftarrow \emptyset$
2: for $i \in [1, |\overline{\sigma}|]$ do
3:     for $j \in [i, |\overline{\sigma}|]$ do
4:         $v \leftarrow \nu_\pi(\overline{\sigma}[i..j])$
5:         if $v = \top$ then
6:             $\mathcal{M} \leftarrow \mathcal{M} \cup \{[i, j]\}$
7:             break
8:         end if
9:     end for
10: end for
11: return $\mathcal{M}$
```

This mechanism presents the advantage of being both simple and very generic. It can detect all occurrences of any pattern, provided that it can be expressed as the set of sequences which produce a match for some monitor $\pi$. Consider, for example, the monitor $\pi_4$ defined as $\pi_4(\overline{\sigma} \cdot \sigma) = \pi_4(\overline{\sigma}) \cdot \top$ if $\overline{\sigma}[1] = a$ and there exists $2 \leq j \leq |\overline{\sigma} \cdot \sigma|$ such that $(\overline{\sigma} \cdot \sigma)[i] = b$; the monitor returns $\bot^{|\overline{\sigma} \cdot \sigma|}$ if $\overline{\sigma}[1] \neq a$ and $?^{|\overline{\sigma} \cdot \sigma|}$ otherwise. In other words, this monitor identifies as a match any stream that starts with an $a$ and contains at some point later at least one occurrence of $b$. Running Algorithm 1 using $\pi_4$ as the monitor and $\overline{\sigma} = bacbacacba$ as the input trace produces the output set $\{[2, 4], [5, 9], [7, 9]\}$. Indeed, one can observe that, for this monitor, those three ranges correspond to the three suffix-minimal matching subsequences of $\overline{\sigma}$.

However, even though simple and generic, this procedure suffers from two important drawbacks. First of all, it is wasteful in resources: upon receiving the $n$-th input event, $n$ instances of $\pi$ must be evaluated on suffixes of length ranging from 1 to $n$. The process repeats on each input event, resulting in a cumulative total of $O(n^3)$ events being ingested by some monitor instance at the $n$-th step. This further assumes that the processing of each event by a monitor is constant in the length of the input, which may not be the case. A first observation allows us to spare a few evaluations of $\pi$ under some circumstances. Note that by the definition of a monitor, if $\overline{\sigma}$ is a non-match, then so will be all its extensions. It is therefore useless to keep evaluating $\pi$ on extensions of a sequence that is already known to be a non-match.

A second drawback comes from the fact that the procedure only identifies for each match a range of events. Yet, it may often be the case that not all events inside that range are actually relevant for the match in question. Consider, for instance, the match corresponding to the range $[5, 9]$ in the previous example. This corresponds to the subsequence *acacb*: it does start with an $a$ and ends with a $b$, but the events that occur in between have no impact on the fact that this subsequence is match.

For this simple example, it is easy to understand that only the start and end of the range are relevant, but making the same kind of judgment for patterns of arbitrary complexity is far from trivial.

## 9.4 State-Based Simplifications

The previous section framed the problem of pattern detection on event streams as a monitoring problem and provided an arguably naïve procedure for identifying all instances of suffix-minimal matches of an arbitrary pattern expressed by some monitor. Yet, to make the problem actually tractable in real-world situations, additional refinements to this basic procedure need to be devised. In addition, it would be desirable to devise ways to single out the events of a range that somehow "explain" the occurrence of the match and doing so in an automated fashion. To address these issues, this section introduces the notion of processor *state* and shows how this concept can be used both to optimize the pattern detection procedure and also identify relevant events of a match.

### 9.4.1 Processor State

Formally, for a processor $\pi : \Sigma^* \to \Sigma'^*$, a function $\iota_\pi : \Sigma^* \to S$ is said to be a *state function* for some arbitrary set $S$ if for every $\overline{\sigma}_1, \overline{\sigma}_2 \in \Sigma^*$, $\iota_\pi(\overline{\sigma}_1) = \iota_\pi(\overline{\sigma}_2)$ implies $\pi(\overline{\sigma}_1 \cdot \overline{\sigma}') = \pi(\overline{\sigma}_2 \cdot \overline{\sigma}')$ for every $\overline{\sigma}' \in \Sigma^*$. The intuition behind this definition is that an element $s \in S$ represents the current "internal state" of $\pi$ after reading a prefix of a sequence of events. In addition, if two prefixes are such that $\pi$ ends in the same state after reading them, extending each of these two prefixes in the same way results in the same output. In other words, the output depends on no other external parameter.

It should be observed that the definition of state does not impose that two processors reaching the same state remain in matching states for any extension of their respective input stream. That is, if $\iota_\pi(\overline{\sigma}_1) = \iota_\pi(\overline{\sigma}_2)$, we allow states $s_1 = \iota_\pi(\overline{\sigma}_1 \cdot \overline{\sigma}')$ and $s_2 = \iota_\pi(\overline{\sigma}_2 \cdot \overline{\sigma}')$ to be different. However, by virtue of the condition stated above, $s_1$ and $s_2$ must be indistinguishable as far as the processor's output is concerned and can therefore be considered as duplicates.

Note that any processor $\pi$ admits a trivial state function $\iota_\pi$ by setting $S = \Sigma^*$ and defining $\iota_\pi(\overline{\sigma}) = \overline{\sigma}$ for all $\overline{\sigma} \in \Sigma^*$. We say that a state function $\iota'_\pi : \Sigma^* \to S'$ is a *contraction* of $\iota_\pi$ if there exists a mapping $\mu : S \to S'$ such that $\iota'_\pi(\overline{\sigma}) = \mu(\iota_\pi(\overline{\sigma}))$ for every $\overline{\sigma} \in \Sigma^*$. The contraction is strict if $\mu$ is not injective. In such a case, $\iota'_\pi$ defines a "tighter" set of states than $\iota_\pi$. A state function is optimal for a processor $\pi$ if it cannot be further contracted. A processor is called *finite-state* when its optimal state function $\iota_\pi : \Sigma^* \to S$ is such that $S$ is finite and *stateless* if $|S| = 1$.[1] The state $s$ such that $s = \iota_\pi(\epsilon)$ is called the initial state of $\pi$.

As a simple example, consider the monitor $\pi_1$ defined in Sect. 9.3.2. Let $S = \{0, 1, 2\}$ and $\iota_{\pi_1}$ be defined as follows:

$$\iota_{\pi_1}(\overline{\sigma}) \triangleq \begin{cases} 0 & \text{if } \overline{\sigma} = \epsilon \text{ or } \overline{\sigma} = a \\ 1 & \text{if } \overline{\sigma}[1..2] = ab \\ 2 & \text{otherwise} \end{cases}$$

One can observe that $\iota_{\pi_1}$ fulfills the conditions of a state function for $\pi_1$. The symbols used for each state are arbitrary; however, intuition shows that, in this particular case, state 0 corresponds to the situation of either an empty stream or the single $a$ event, state 1 corresponds to the state of the monitor after having seen $ab$ as the first two events of a stream, and state 2 corresponds to a non-match.

Although the states in this example match the three possible verdicts produced by the monitor, this is not always the case. Consider as a second example monitor $\pi_4$. One can let $S = \mathbb{N} \times \mathbb{N} \times \mathbb{B}_3$ and define a state function $\iota_{\pi_4}(\overline{\sigma}) \triangleq (|\overline{\sigma}|_a, |\overline{\sigma}|_b, \pi_4(\overline{\sigma}))$. This time, the state of the monitor after ingesting a stream $\overline{\sigma}$ is made of the number of symbols $a$ and $b$ present in $\overline{\sigma}$, as well as a flag indicating if a match or non-match has already been declared. On can indeed observe that knowledge of these three values suffices to determine the output of $\pi_4$ for any extension of that stream.[2] Note however that this definition does not result in an optimal state function: for instance, any pair of states $(x, y, \bot)$ and $(x', y', \bot)$ result in the same output for any extension (namely, the verdict $\bot$) despite being distinct states.

The interest of this definition of state function is that, despite its name, it does not imply that its corresponding processor $\pi$ be expressed as a form of state machine. As a matter of fact, $S$ can even be infinite. Of course, when such a notation is used, what constitutes a possible state function is obvious; however, a state function can also be constructed for arbitrary processors, as the previous example has shown.

Given a state function $\iota_\pi$, we shall designate the fact that a processor $\pi$ is currently in state $s$ by the notation $\pi^s$. This processor differs from $\pi$ as it handles events from its current state, which may not be the initial state. Formally, this means that for every $\overline{\sigma} \in \Sigma^*$:

$$\pi^s(\overline{\sigma}) = \pi(\overline{\sigma}' \cdot \overline{\sigma})[|\pi(\overline{\sigma})|..]$$

[1] The term "stateless" may seem odd since $\iota_\pi$ actually has one state; however, this means that the output of $\pi$ does not depend on its internal state, precisely since it is always in the same state.

[2] As a matter of fact, neither of these three elements could be taken out without violating the condition for a state function. In particular, knowledge of $|\overline{\sigma}|_a$ and $|\overline{\sigma}|_b$ is not sufficient to determine the monitor's verdict, as the order in which the symbols occur may or may not result in a prefix of $\overline{\sigma}$ satisfying the condition.

for any $\overline{\sigma}' \in \Sigma^*$ such that $\iota_\pi(\overline{\sigma}') = s$. The definition of a state function ensures us that the choice of $\overline{\sigma}'$ is arbitrary. In other words, evaluating $\pi^s$ on an input stream $\overline{\sigma}$ is equivalent to finding a stream $\overline{\sigma}$ that takes $\pi$ from its initial state to state $s$, evaluating $\pi(\overline{\sigma}' \cdot \overline{\sigma})$, and then trimming whatever events are produced by evaluating $\pi$ on the prefix $\overline{\sigma}$.

**Algorithm 2** State-based search for suffix-minimal matches in $\overline{\sigma}$ for a monitor $\pi$

```
1:  $\mathcal{M} \leftarrow \emptyset, \overline{\pi} \leftarrow \epsilon, \Delta \leftarrow \emptyset$
2:  for $j \in [1, |\overline{\sigma}|]$ do
3:      $\overline{\pi} \leftarrow \overline{\pi} \cdot \pi^{\iota_\pi(\epsilon)}$
4:      for $i \in [1, |\overline{\pi}|]$ do
5:          if $i \in \Delta$ then skip
6:          $S \leftarrow \emptyset$
7:          $\pi^s \leftarrow \overline{\pi}[i]$
8:          $v \leftarrow \nu_{\pi^s}(\overline{\sigma}[j])$
9:          $s' \leftarrow \iota_{\pi^s}(\overline{\sigma}[j])$
10:         if $s = s' = \iota_\pi(\epsilon)$ or $s' \in S$ then
11:             $\Delta \leftarrow \Delta \cup \{j\}$
12:             skip
13:         end if
14:         $S \leftarrow S \cup \{s'\}$
15:         if $v = \top$ then
16:             $\mathcal{M} \leftarrow \mathcal{M} \cup \{[i, j]\}$
17:             $\Delta \leftarrow \Delta \cup \{j\}$
18:         end if
19:         if $v = \bot$ then
20:             $\Delta \leftarrow \Delta \cup \{j\}$
21:         end if
22:         $\overline{\pi}[j] \leftarrow \pi^{s'}$
23:     end for
24: end for
25: return $\mathcal{M}$
```

This notion of state can help us further trim the set of relevant matches (and corresponding monitor evaluations) that need to be handled. Consider, for example, the pattern stipulating that a match is a sequence where some $b$ is immediately followed by a $c$. The sequence $\overline{\sigma} = aaabc$ is a match and even a minimal match of $\pi$. However, remark that the first three $a$ events do not really matter; they can be seen as some "filler" that are not material witnesses that the sequence satisfies the condition. As a matter of fact, $aabc$, $abc$, and $bc$, which are all suffixes of $\overline{\sigma}$, also are matches.

This behavior can be tied to a property of the monitor's state when ingesting the input sequence. Based on the definition of $\pi$ in this example, we can easily see that for every trace $\overline{\sigma}$, $\pi(\overline{\sigma}) = \pi(a \cdot \overline{\sigma})$. In other words, if $\overline{\sigma}$ is a match, prepending an $a$ at the beginning of $\overline{\sigma}$ is also a match (and the same for non-matches and potential matches). Setting $\overline{\sigma} = \epsilon$, we conclude that $\iota_\pi(\epsilon) = \iota_\pi(a)$: thus, for any number of $a$ at the beginning of a sequence, $\pi$ remains in the same internal state – its initial state. This explains why intuition tells us that these first events are useless to assess whether the sequence matches the condition: the monitor itself acts as if they were absent from the trace.

### 9.4.2 A State-Aware Detection Algorithm

Additional reductions on the number of potential matches can be obtained by generalizing this principle one step further and considering as duplicates any matches placing their respective monitor into the same state for a given position in the stream. Consider, for example, the monitor $\pi_5$ defined as follows:

$$\pi_5(\overline{\sigma} \cdot \sigma) = \begin{cases} \pi_5(\overline{\sigma}) \cdot \top & \text{if } \exists\ 1 \leq i < j < k \leq |\overline{\sigma}| \text{ such that} \\ & (\overline{\sigma} \cdot \sigma)[i] = a, (\overline{\sigma} \cdot \sigma)[j] = b \text{ and } (\overline{\sigma} \cdot \sigma)[k] = c \\ \pi_5(\overline{\sigma}) \cdot ? & \text{otherwise} \end{cases}$$

This monitor declares a match whenever the stream contains the succession of events $a$, followed some time later by $b$, followed some time later by $c$. A state function $\iota_{\pi_5}$ can be devised with four states $\{0, 1, 2, 3\}$, where $\iota_{\pi_5}(\overline{\sigma}) = 0$ for all streams containing neither $a$ nor $b$ nor $c$, $\iota_{\pi_5}(\overline{\sigma}) = 1$ for streams where no occurrence of $a$ is followed by $b$, $\iota_{\pi_5}(\overline{\sigma}) = 2$ for streams where no occurrence of $a$ followed by $b$ is followed by $c$, and $\iota_{\pi_5}(\overline{\sigma}) = 3$ for streams containing the desired pattern.

Let $\overline{\sigma} = ababacbc$ be the stream on which this pattern is to be detected. Applying Algorithm 1 to it results in the set of matches $\{[1, 6], [3, 6], [5, 8]\}$. However, remark that a monitor reading the first four events (i.e., $\overline{\sigma}[1..4]$) ends up in the same state as the monitor reading only the third and fourth (i.e., $\overline{\sigma}[3..4]$), as, in both cases, an $a$ followed by a $b$ has been observed, thus resulting in state 2. From that point on, any suffix causing a match for the first monitor will also cause a match for the second. One could thus consider that these two matches are redundant and only report one of them instead of both.

These various observations yield an improved technique for detecting matches, which is described in Algorithm 2. On the first line, the algorithm initializes and later maintains three data structures: the set of found matches $\mathcal{M}$, a list of monitor instances $\overline{\pi}$, and a set $\Delta$ that will contain the indices of $\overline{\pi}$ corresponding to monitor instances that no longer need to be considered. The algorithm iterates over each event of the input stream $\overline{\sigma}$; line 3 appends at the end of $\overline{\pi}$ a new fresh instance of the monitor $\pi$ in its initial state. Then, on lines 4–23, the same process is repeated for each monitor instance contained in the list $\overline{\pi}$. First, the monitor and its current state $\pi^s$ are retrieved from the list (line 7); this monitor is then fed the current event from the stream, and its verdict and new state are obtained (lines 8–9). At the end of the iteration, the monitor in its new state overwrites the original monitor at the corresponding position in $\overline{\pi}$ (line 22).

As a result, at the end of the $j$-th iteration of the loop in line 2, $\overline{\pi}$ contains $j$ distinct instances of the monitor $\pi$, and the $i$-th element of this list is a monitor $\pi^s$ such that $s = \iota_\pi(\overline{\sigma}[i..j])$. That is, the first monitor instance reads the input stream from the first event, the second reads it from the second event, and so on. Each of these monitors is fed one more event from the stream for each iteration of the inner loop. Any monitor instance whose index ends up in the set $\Delta$ is considered "done" and is not handled anymore, as is represented by the skip condition of line 5. On each iteration of the inner loop, a set of states $S$ is initialized, and the state of each monitor instance after ingesting the current event is stored in this set (line 14).

Lines 15–21 take care of the various situations that can occur depending on a monitor's verdict. Lines 15–18 handle the case where the monitor declares a match; in such a case, the range of events of the input stream consumed by this monitor instance is added to the set of matches $\mathcal{M}$, and its position in $\overline{\pi}$ is added to the set $\Delta$ of indices to discard. Lines 19–21 perform a similar task for the case of a non-match. Finally, lines 10–13 implement the simplifications discussed earlier based on the monitor's state. If a monitor in its initial state remains in its initial state after ingesting the current event or if its new state is identical to the state of another monitor processed in the same iteration, its position in $\overline{\pi}$ is also added to the set $\Delta$ of indices to discard.

One can observe that the set of $\mathcal{M}$ produced by Algorithm 2 is the subset of $\mathcal{M}_\pi$ containing only suffix-minimal matches that take the monitor out of its initial state on ingesting the first event and that are such that no two matches result in monitors having the same state at the same position in the stream.

## *9.4.3 Progressing Subsequences*

Algorithm 2 presents several improvements over the naïve procedure introduced in Algorithm 1: First, it reduces the total number of events ingested by monitor instances: every time a new input event is to be processed, existing instances are only fed this new event from their existing state, instead of re-evaluating the sub-trace from the start. This, in itself, reduces its complexity to $O(n^2)$. In addition, it implements mechanisms to reduce the number of "live" monitor instances that need to be handled at any point in time and consequently reduces the number of (essentially redundant) matches produced by Algorithm 1. It does so at the price of memory, as in the worst case, consuming $n$ input events may necessitate to keep $n$ distinct monitor instances along with their internal state.[3]

However, these optimizations still do not address the second concern that was expressed for Algorithm 1, namely, that only ranges of events for each match are provided, regardless of whether these events are all actually relevant to the

[3] This worst case is arguably contrived, as it would require each successive event of the stream to place the corresponding new monitor instance in a different state as that of all previous monitors.

identification of the pattern. To this end, the state function associated with a processor can be put to another use, which we describe in the following:

Let $\pi$ be a processor with associated state function $\iota_\pi$ and $\overline{\sigma} \in \Sigma^*$ be a sequence of events. The *state sequence* of $\overline{\sigma}$ is the sequence $\overline{s}$ such that $\overline{s}[0] = \iota_\pi(\epsilon)$ and $\overline{s}[i+1] = \iota_\pi(\overline{\sigma}[1..i])$ for every $i \geq 0$. Thus, the state sequence starts with the initial state of $\pi$, which is followed by the states reached by $\pi$ after ingesting each successive event of $\overline{\sigma}$. Let $\overline{\sigma}'$ be a subsequence of $\overline{\sigma}$; we likewise define $\overline{s}'$ as the state sequence of $\overline{\sigma}'$. We say that $\overline{\sigma}'$ is the progressing subsequence of $\overline{\sigma}$ if $\overline{s}'$ is the result of removing all loops from $\overline{s}$.

Equipped with this definition, we can return to the example of monitor $\pi_5$. Assuming without loss of generality that $\pi$ starts in state $s_0$ and moves to state $s_1$ when reading $b$ and to $s_2$ when reading $c$ afterward, the state sequence for the input $aaabc$ is $s_0s_0s_0s_0s_1s_2$. The sequence $bc$, on its side, results in the state sequence $s_0s_1s_2$, which is exactly the result of removing all loops from the state sequence of the original input. Thus, the intuition that the first $a$ events are irrelevant to $\pi$ becomes a consequence of the definition of progressive subsequence.

For processors expressed as a finite-state machine, this definition is relatively easy to illustrate. Consider the Moore machine $\pi.$ shown in Fig. 9.4, which associates with each state a verdict in $\mathbb{B}_3$. The sequence $aabcbcda$ ends in the state 4, labeled with $\top$; hence, it is a match according to $\pi$. Its corresponding progressing subsequence is $abd$. One can observe, and it can easily be shown, that state $s_1$ is visited before $s_2$ in the original sequence if and only if it is also visited before in the progressing subsequence.

The definition allows portions of the original sequence to be deleted, but not just anyhow: only loops are removed, but the remainder of the path remains untouched. In particular, a progressing subsequence is not in general the shortest subsequence ending in the final state. Thus, in the example above, the sequence $abcd$ leads to state 5, and $ad$ is its progressing subsequence. However, $d$ is a subsequence of $abcd$ that would directly lead to state 5, but it is not a *progressing* subsequence.

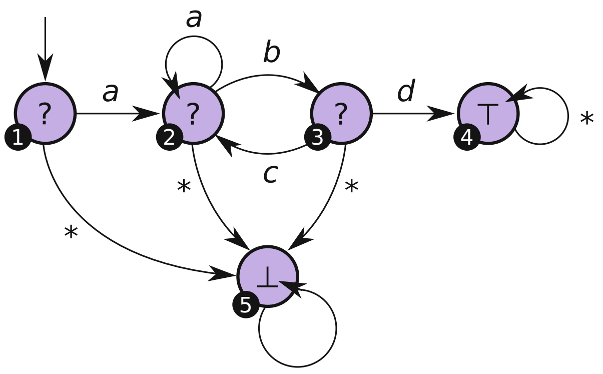


**Fig. 9.4** A simple Moore machine defining valid sequences of atomic events

The notion of progressive subsequence is a generalization of one of the simplification strategies that were already considered in the procedure proposed in Sect. 9.4.1. Given an existing sequence $\overline{\sigma}$ of length $n \geq 0$ and upon receiving a new event $\sigma$, if $\pi(\sigma) = \pi(\epsilon)$, then $\sigma$ may be the start of a match but, as per the definition above, is not the start of a *progressive* subsequence of a potential match. Thus, Algorithm 2 can be further improved by altering line 16: instead of giving the complete range $[i, j]$ as the witness of the match for a monitor, one can instead only provide the subset of this interval corresponding to the progressing subsequence as defined above.

## 9.4.4 Combining Reduction Strategies

Figure 9.5 shows by a simple example the difference in the operation of each detection strategy described above. The pattern to be detected is an event $A$, eventually followed by a $B$, eventually followed by a $C$, with arbitrary interleaving events between each of them. The approach called "Direct" is the one we described first; it consists of starting a new monitor instance on each new input event and reporting as occurrences of the pattern the complete sequence of events ingested by each monitor that declares a match. In the simple trace given as an example, this results in the detection of eight occurrences of the pattern, with the events contained in each pattern instance being marked by a $\times$ symbol.

The "first step" detection strategy prunes from these matches any monitor instance that does not change its state upon ingesting its first event. As per the

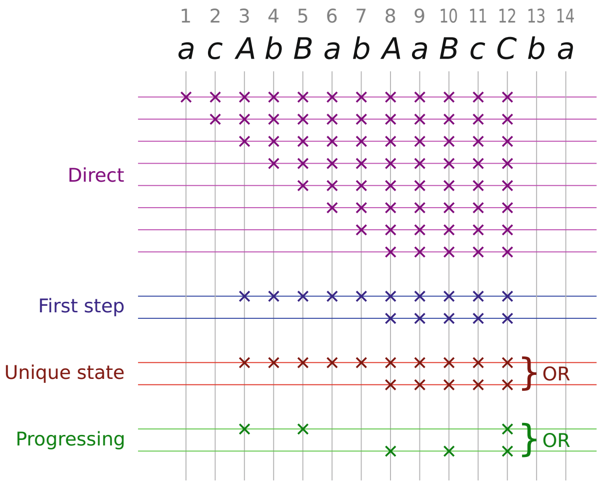


**Fig. 9.5** A comparison of the four monitor-based detection strategies on a simple trace where the pattern $A \rightarrow B \rightarrow C$ is to be detected

definitions above, in the present case, the first state change occurs only in the presence of an $A$. Thus, only two of the eight pattern instances remain. Note however that the events included in each of these patterns still contain the complete sequence processed by each monitor instance up until a match is found.

The “unique state” strategy prunes from the active monitors any instance that reaches the same state as another monitor on a given input position. This is the case here upon the ingestion of event $B$ at position 10. At this point, two monitor instances end up in the state corresponding to having read an $A$, followed by a $B$: the first instance was started at position 3 and the second at position 8. According to the unique state strategy, one of these monitor instances can be discarded; thus, a single of the two pattern matches will be reported. As discussed earlier, various criteria can be applied to decide which, among the multiple monitor instances, is to be kept.

Finally, the last reduction strategy, “progressing,” applies further reductions on the portions of the input trace that are part of a match. Note this time that the number of matches itself is left unchanged. However, in each instance, only the events that are part of the progressing subsequence are retained. These results are arguably closer to intuition, as, in each instance, only the respective positions of the $A$, $B$, and $C$ events are kept. Also note that each pattern match contains only *one* of each event; for instance, even though the first pattern instance spans the events from position 3 to 12, event $A$ at position 8 is not included.

The cumulative application of these reduction strategies results in a sharp decrease in both the number of reported matches and the number of events involved as witnesses of each match. As one can see, the direct approach results in 8 mostly redundant matches, including all but 2 of the 14 events of the input. In contrast, the progressing strategy reports a single match containing only three events of the input. This potential for reduction will be examined further in Sect. 9.6.

## 9.5 A Compositional Approach to Pattern Detection

Although monitors represent a powerful method for expressing and detecting patterns in a stream of events, defining each such pattern using a custom-made monitor is not feasible in practice. It is hard to imagine a setup where each new monitor would need to be defined from scratch as a function, which would probably be much more complex than the simple examples used so far in this chapter. What is more, for any of the optimizations introduced in Sect. 9.4 to be applicable, one must also devise an appropriate notion of processor state for each monitor created in such a way.

However, elaborate relationships between events in an input sequence can be captured by *composing* processors together; in such a setting, the output of a processor is given as the input of another one, forming potentially complex graphs where events of the original stream are progressively transformed into the required $\top/?/\bot$ verdict. This is the approach proposed in this section. Instead of requiring monitors to be defined directly as big, monolithic functions, we suggest a number of elementary "building blocks," which can then be composed to represent the desired patterns to be detected.

The main advantage of such an approach is that the definition of a state function and the extraction of progressing subsequences for these pipelines essentially come "for free." As we shall see, if each elementary processor has its own state function and can identify its progressing subsequences, calculating the subsequence of the end-to-end chain can be done by "composing" these individual subsequences, thus sparing a user from calculating these manually.

### *9.5.1 Building Blocks for Pattern Detection*

We start the section by introducing and formally defining a number of generic and elementary processors that can be used to detect various types of pattern. The presentation is divided into a set of processors performing generic manipulations on event streams (not necessarily tied to monitoring), followed by a set of monitors specific to the detection of patterns in event streams.

Since compositions of processors are best represented graphically, we shall associate with each of them a pictogram representing its function. The convention we use is to represent processors as square boxes, with input and output "pipes" designating each of the streams that are consumed or produced by the processor. The use of colors for pipes helps distinguish the type of events in the corresponding stream; in the following, pink represents atomic events from an arbitrary alphabet $\Sigma$, dark green indicates numbers, blue indicates Boolean values, while purple corresponds to ternary Boolean values ($\mathbb{B}_3$). A white pipe represents an arbitrary type.

#### **9.5.1.1 Generic Processors**

For the set of core processors, we hitchhike on past works on the topic and reuse those introduced by Bédard and Hallé [10]; for this reason, we shall only briefly present and define each of these core processors, whose graphical representation is shown in Fig. 9.6.

First, the *Fork* processor is merely a structural construct allowing a single stream to be duplicated and sent as the input of multiple processors: $\pi(\boldsymbol{\sigma}) \triangleq \langle \boldsymbol{\sigma}, \dots, \boldsymbol{\sigma} \rangle$. The *ApplyFunction* processor lifts any function $f : \Sigma_1 \times \cdots \times \Sigma_m \rightarrow \Sigma'_1 \times \cdots \times \Sigma'_n$ into a processor $\pi : (\Sigma_1 \times \cdots \times \Sigma_m)^* \rightarrow (\Sigma'_1 \times \cdots \times \Sigma'_n)^*$ defined as $\pi(\boldsymbol{\sigma} \cdot (\sigma_1, \dots, \sigma_m)) \triangleq \pi(\boldsymbol{\sigma}) \cdot f(\sigma_1, \dots, \sigma_m)$.[4] *CountDecimate* is a 1:1 processor

[4] Note that the function produces exactly one output front for each input front; thus, it cannot insert or delete events like some other processors.

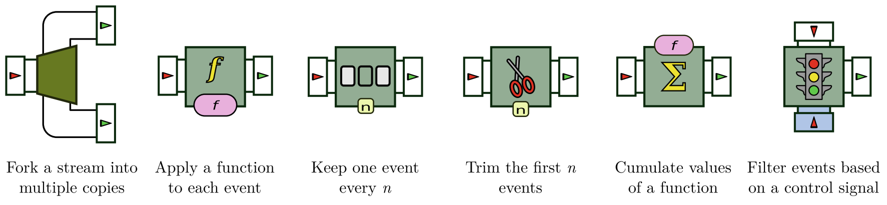

Fork a stream into multiple copies
Apply a function to each event
Keep one event every $n$
Trim the first $n$ events
Cumulate values of a function
Filter events based on a control signal

**Fig. 9.6** Pictorial representation of generic processors for stream manipulation

that keeps one event every $k$ and is defined as $\pi(\boldsymbol{\sigma}) \triangleq \langle \boldsymbol{\sigma}[0], \boldsymbol{\sigma}[k], \boldsymbol{\sigma}[2k], \dots \rangle$. *Trim* removes the first $k$ events of the stream and is defined as $\pi(\boldsymbol{\sigma}) \triangleq \langle \boldsymbol{\sigma}[k], \boldsymbol{\sigma}[k+1], \boldsymbol{\sigma}[k+2], \dots \rangle$. *Filter* is a processor $\pi : (\Sigma \times \{\top, \bot\})^* \to \Sigma^*$ that discards events based on a stream of Boolean values. The event at position $n$ in the first stream is sent to the output if and only if the event at the same position in the second stream is the Boolean value true; formally, $\pi(\boldsymbol{\sigma} \cdot (\sigma, b)) \triangleq \pi(\boldsymbol{\sigma}) \cdot \sigma$ if $b = \top$ and $\pi(\boldsymbol{\sigma})$ otherwise.

As its name implies, the *Cumulate* processor is designed to "accumulate" the successive values of a binary function. Given a function $f : \Sigma^2 \to \Sigma$ and an implicit initial value $\sigma_0 \in \Sigma$, the processor is defined recursively as $\pi(\langle\sigma\rangle) \triangleq \langle f(\sigma_0, \sigma)\rangle$ and $\pi(\langle\overline{\sigma}\cdot\sigma\rangle) = \pi(\langle\overline{\sigma}\rangle)\cdot\langle f(\pi(\langle\overline{\sigma}\rangle)[-1], \sigma)\rangle$, where $\pi(\langle\overline{\sigma}\rangle)[-1]$ stands for the last event produced by $\pi$ on the input stream $\overline{\sigma}$. This generic construction can represent various types of computations depending on the function used. For example, if $f$ is addition and $\sigma_0 = 0$ is used as the start value, $\pi$ produces an output stream where the $i$-th event is the sum of all input events up to the $i$-th. If $f$ is Boolean conjunction and $\sigma_0 = \top$, $\pi$ produces an output stream where the $i$-the event is the conjunction of all input events up to the $i$-th.

### 9.5.1.2 Elementary Monitors

We continue by introducing a handful of new elementary monitors that complement the core processors and which are especially suitable for the task of pattern detection in event streams. Their graphical representation is shown in Fig. 9.7. To distinguish these monitors from more generic processors, they are represented as boxes with wedged corners.

A first monitor is called *Sequence*, which is defined as follows:

$$\pi_{,}((\sigma_0, \sigma'_0), \dots, (\sigma_k, \sigma'_k)) \triangleq \begin{cases} \pi_{,}((\sigma_0, \sigma'_0), \dots, (\sigma_{k-1}, \sigma'_{k-1})) \cdot \top \\ \qquad \text{if } \exists\, 1 \leq i < j \leq k \text{ s. t. } \sigma_i = \top \text{ and } \sigma_j = \top \\ \pi_{,}((\sigma_0, \sigma'_0), \dots, (\sigma_{k-1}, \sigma'_{k-1})) \cdot ? \text{ otherwise} \end{cases}$$

Note that this monitor receives as input two streams of ternary Boolean values; it produces the verdict $\top$ whenever the first stream contains the value $\top$ and the

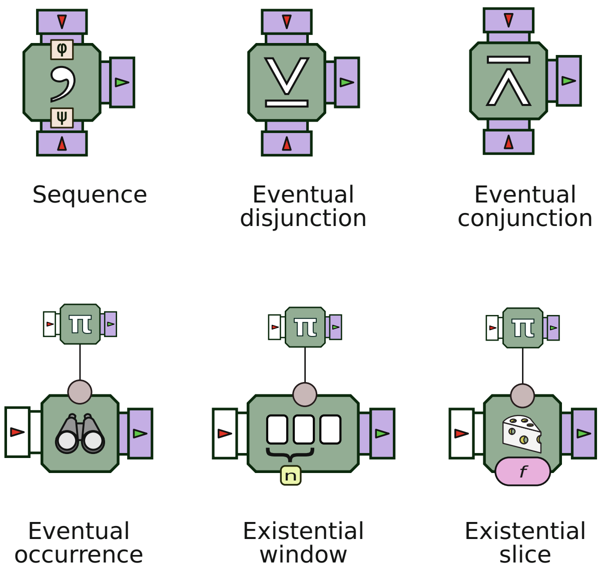


**Fig. 9.7** A basic set of monitors for pattern detection

second stream contains the value ⊤ at a *subsequent* position. If each input stream corresponds to the evaluation of a condition on some other stream, this monitor declares a match when it sees the first condition evaluates to true, followed by the second condition some time later.

The *Eventual disjunction* monitor also receives as input two streams of ternary Boolean values; it produces the verdict ⊤ as soon as any of the two input streams contains the value ⊤; this is formally defined as:

$$\pi_{\veebar}((\sigma_0, \sigma'_0), \ldots, (\sigma_k, \sigma'_k)) \triangleq \begin{cases} \pi_{\veebar}((\sigma_0, \sigma'_0), \ldots, (\sigma_{k-1}, \sigma'_{k-1})) \cdot \top \\ \qquad \text{if } \exists\, 1 \leq i \leq k \text{ s. t. } \sigma_i = \top \text{ or } \sigma'_i = \top \\ \pi_{\veebar}((\sigma_0, \sigma'_0), \ldots, (\sigma_{k-1}, \sigma'_{k-1})) \cdot ? \text{ otherwise} \end{cases}$$

Again, if each input stream given to this monitor corresponds to the evaluation of a condition on some other stream, this monitor declares a match when either of these conditions evaluates to true. The progressing subsequence of a match for this monitor corresponds to the location of the first occurrence of ⊤ in any of the two input streams. The *Eventual conjunction* monitor works dually and declares a match once *both* its input streams contain the value ⊤ (which need not occur at the same position). In such a case, the progressing subsequence of a match for this monitor corresponds to the location of the first occurrence of ⊤ in both input streams.

The *Eventual occurrence* monitor takes as input a single event stream and also requires as a parameter another monitor $\pi$. It spawns one new instance of $\pi$ at each input event and keeps feeding input events to each of these monitor instances until one of them produces the verdict $\top$. This is formally defined as follows:

$$\pi_E(\overline{\sigma}) \triangleq \begin{cases} \pi_E(\overline{\sigma}) \cdot \top & \text{if } \exists\ 1 \leq i \leq k \text{ s. t. } \nu(\pi(\overline{\sigma}[i..k])) = \top \\ \pi_E(\overline{\sigma}) \cdot ? & \text{otherwise} \end{cases}$$

Thus, if $\pi$ is a monitor representing an arbitrary pattern, *Eventual occurrence* declares a match whenever $\pi$ declares a match for a given suffix of the input, thus representing the fact that the pattern eventually occurs at some point down the stream. The progressing subsequence of a match for this monitor corresponds to the progressing subsequence of the instance of $\pi$ that declares a match, offset by the starting position in the stream of this monitor instance.

*Existential window* operates in a similar manner, except that each instance of $\pi$ is evaluated on a window of fixed width $n$, instead of the complete suffix of the input stream:

$$\pi_W(\overline{\sigma}) \triangleq \begin{cases} \pi_W(\overline{\sigma}) \cdot \top & \text{if } \exists\ 1 \leq i \leq k \text{ s. t. } \nu(\pi(\overline{\sigma}[i..k-n])) = \top \\ \pi_W(\overline{\sigma}) \cdot ? & \text{otherwise} \end{cases}$$

It declares a match if there exists an interval of $k$ successive events in the input stream for which $\pi$ declares a match. The progressing subsequence is defined in the same way as for *Eventual occurrence*.

Finally, *Existential slice* is a monitor that separates an input stream into multiple sub-streams called *slices*. The monitor takes as arguments another monitor $\pi$ and a function $f : \Sigma \to C$, producing values in some arbitrary set $C$. For a stream $\overline{\sigma}$, we note as $[\overline{\sigma}]_c^f$ the subsequence of $\overline{\sigma}$ containing only events $\sigma$ such that $f(\sigma) = c$, which is the "slice" corresponding to $c$. The monitor runs one instance of $\pi$ for each value in $C$; on each input event $\sigma$, the value $f(\sigma) = c$ is evaluated; the event is then fed to the processor instance associated with $c$. This processor declares a match whenever one the instances of $\pi$ declares a match. This can be formalized as follows:

$$\pi_S(\overline{\sigma}) \triangleq \begin{cases} \pi_S(\overline{\sigma}) \cdot \top & \text{if } \exists\ c \in C \text{ s. t. } \nu(\pi([\overline{\sigma}]_c^f)) = \top \\ \pi_S(\overline{\sigma}) \cdot ? & \text{otherwise} \end{cases}$$

The progressing subsequence of a match for this monitor corresponds to the progressing subsequence of the underlying monitor instance declaring a match, by taking care of replacing the events of this slice to their actual position in the input stream.

## *9.5.2 Progressive Subsequences for Processor Pipelines*

Each of these processors in itself performs a very simple task. As discussed at the start of this section, complex patterns are not expected to be expressed directly through a single instance of one of these processors but rather as a composition of elementary processors and monitors.

### 9.5.2.1 Pipeline Definition

A *processor pipeline* is defined as a tuple $\mathcal{P} = \langle \overline{P}, E \rangle$, where $\overline{P} \in \Pi^*$ is a list of processors and $E \subseteq \mathbb{N}^4$ is a list of edges. An element $(p, n, p', n')$ of $E$ stipulates that the $n$-th output stream of processor $\overline{P}[p]$ is set to be the $n'$-th input of processor $\overline{P}[p']$. One can see a pipeline as a graph where vertices are processors with upstream and downstream "pipes" (input and output streams) and edges connect downstream pipes to upstream pipes.

Figure 9.8 shows a graphical representation of a simple processor pipeline, making use of some of the processors introduced earlier. A line between two pipes represents a (directed) connection. For processors with more than one input or output, pipes are ordered from top to bottom by convention and a symbol or a number by be affixed to them to avoid confusion. This computational model is reminiscent of the "data flows" presented by Woodruff et al. [58]. However, whereas data flows are tuple-oriented, our proposed computational model is more generic and accommodates arbitrary processors and data types.

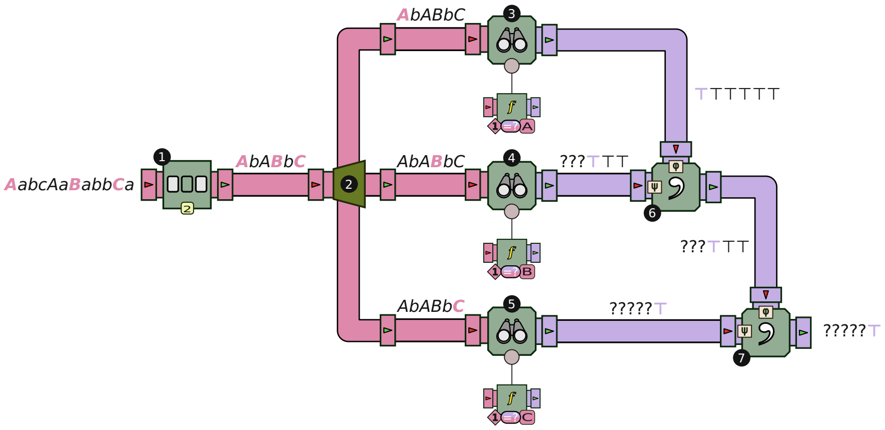


**Fig. 9.8** A monitor identifying a simple linear pattern formed of a sequence of $A$, $B$, and $C$, interleaved with an arbitrary number of other events. The match is declared only if these events are observed at odd indices in the stream. Next to each pipe are the contents of each stream for a possible input

The operation of this pipeline can be analyzed to understand the pattern that this monitor can catch. At the left of the pipeline, the stream of input events is first decimated (box #1), so that every other event from the input stream is discarded. This entails that the remainder of the processing is done on a stream composed only of the events from the input that appear at odd positions (1, 3, 5, etc.). This stream is forked into three copies (box #2); on each copy, the eventual occurrence of a different condition is evaluated. Box #3 declares a match when it encounters the first $A$ symbol; box #4 does the same for the symbol $B$, and box #5 looks for an occurrence of symbol $C$.

The output of monitors #3 and #4 is sent into an instance of the *Sequence* monitor (#6): as per the definition introduced earlier, this entails that it declares a match when a symbol $B$ is seen after a symbol $A$. The output of this monitor is plugged into another instance of *Sequence*, along with the output of monitor #5: therefore, it declares a match when a symbol $C$ is seen after an instance of the pattern observed by box #6. The end result of this pipeline is a monitor that looks for a sequence of A, B, and C, interleaved with an arbitrary number of other events; however, because of the presence of the *CountDecimate* processor at the beginning, the match is declared only if these events are observed at odd indices in the stream.

This abstract example shows how a combination of elementary processors and monitors in a pipeline can be used to express complex relationships between events forming a potential pattern. However, it remains to show how, from a match identified by this monitor, one can extract the relevant events of the input stream that are witnesses of the presence of this match.

#### 9.5.2.2 Input-Output Associations

For each processor $\pi : (\Sigma_1 \times \cdots \times \Sigma_m)^* \rightarrow (\Sigma'_1 \times \cdots \times \Sigma'_n)^*$, we introduce a function $\rho_\pi : (\Sigma_1 \times \cdots \times \Sigma_m)^* \times \mathbb{N} \times \mathbb{N} \rightarrow 2^{\mathbb{N}\times\mathbb{N}}$ that associates output events with input events given to a processor. More precisely, given an input stream vector $\boldsymbol{\sigma} \in (\Sigma_1 \times \cdots \times \Sigma_m)^*$, the index of an output stream $i \in [1, n]$, and the position $x \in [1, ||\pi(\boldsymbol{\sigma})||]$ of an event inside that stream, $\rho_\pi(\boldsymbol{\sigma}, i, x)$ produces a set of the form $\{(j_1, y_1), \ldots, (j_k, y_k)\}$. A tuple $(j, y)$ of this set is such that $j \in [1, m]$ and $y \in [1, ||\boldsymbol{\sigma}||$. It indicates that the event at position $y$ in the $j$-th input stream given to $\pi$ is part of the evidence explaining the production of the event at position $x$ in the $i$-th output stream. We shall also extend the notation and define $\rho_\pi$ for a set of output event positions $I = \{(i_1, x_1) \ldots, (i_k, x_k)\}$ as:

$$\rho_\pi(\boldsymbol{\sigma}, I) \triangleq \bigcup_{(i,x)\in I} \rho_\pi(\boldsymbol{\sigma}, i, x)$$

Thus, if one designates multiple output events, $\rho$ simply returns the union of all input events associated with any of these output events.

There exist multiple ways of deciding what input events should be associated with a given output event; however, the definition of $\rho$ we propose in this work is nothing but the progressing subsequence induced by the input stream and resulting in the given output event being produced. For example, given the monitor displayed on Fig. 9.4 and the input stream $\overline{\sigma} = aabcbcda$, one obtains as output the stream $??????\top\top$. The events associated with the event at position 7 of the first (and only) output stream of this processor can be obtained by evaluating $\rho_\pi(\overline{\sigma}, 1, 7)$, which, in this case, produces the set of tuples $\{(1, 1), (1, 5), (1, 7)\}$. This links the output event to the first, fifth, and seventh input events of the first (and only) input stream, which, as one can observe, corresponds to the progressing subsequence *abd* we discussed earlier.

Equipped with this function, it becomes straightforward to retrace the relationship between any output event of a chain of composed processors $\pi_1 \circ \cdots \circ \pi_k$ and the positions of events passed to the input of the chain. One simply starts from a given event position $j$ in the output of the last processor $\pi_k$ and uses $\rho_{\pi_k}$ to obtain the set of input events $I$ consumed to produce it. It is then possible to move the next-to-last processor of the chain, $\pi_{k-1}$, and repeat the process by evaluating $\rho_{\pi_{k-1}}$ on the interval $I$. The end result, when reaching the first processor of the chain $\pi_1$, is the set of positions of all events in the original input sequence that are linked to the discovery of a match at position $j$ all the way down the chain.

We can now revisit the example presented above and apply this reasoning on the pipeline of Fig. 9.8. Consider the input sequence $\overline{\sigma} = AabcAaBabbCa$. The illustration shows how this original input stream is transformed into other streams as the events pass through each processor, which eventually leads to the monitor in box #7 producing for its output the sequence $?????\top$: as expected, it declares a match when event $C$ at position 12 is observed.[5] The input given to this processor $\pi$ is the stream vector $\boldsymbol{\sigma} = ((???\top\top\top), (?????\top))$; calculating $\rho_\pi(\boldsymbol{\sigma}, 1, 6)$ amounts to asking what are the events of the progressing subsequence of $\pi$ explaining the production of the verdict $\top$ at the sixth position of its output. According to the definitions introduced earlier, this output event is associated with the event at position 4 in the first input stream and the event at position 6 in the second, thus producing the set $\{(1, 4), (2, 6)\}$. These events are highlighted in Fig. 9.8. We can observe that these tuples correspond to the first occurrence of $\top$ in either stream, consistent with the definition of a progressive subsequence of the *Sequence* processor.

The process can then be repeated by successively pointing at each of these events and calculating the input events that the upstream processor associates with them; these events are highlighted throughout Fig. 9.8. The operation ends at the leftmost processor, which keeps track of the location of output events with respect to their original position in the global input stream. The end result, in this particular example, precisely matches the intuition: the global pipeline looks for a

[5] We remind that because of the presence of box #1, the pipeline only produces one output event for each two input events.

succession of $A$, $B$, and $C$ at odd indices, and the highlighted events are precisely the first occurrence of $A$, $B$, and $C$ that fulfill this condition in the stream. In other words, this shows how the composition of elementary processors and the iterative calculation of individual progressing subsequences do indeed fall back on the appropriate subset of the input explaining the occurrence of a pattern when it is found.

We remind the reader that this process is systematic and automated; given an arbitrary pipeline where each processor present can properly calculate associations between input and output events, the identification of the appropriate subset of the input stream explaining the occurrence of a match requires no manual intervention whatsoever.

## 9.6 Experimental Evaluation

In this last section, we revisit the notions of monitor, state, and progressive subsequence and measure experimentally their impact in terms of computation load and potential for reduction in the number of matches and events in each match. We first describe a concrete implementation of these concepts as an extension of an existing event stream processing library and then report on experimental results obtained by running various monitor pipelines on generated event streams.

### *9.6.1 Implementation*

As the basis of our implementation, we use an actual open-source event stream processing engine, called BeepBeep [30]. BeepBeep offers a collection of simple computation units called `Processors`, which correspond exactly to the definition we gave of this concept in Sect. 9.3.1. Processors can then be connected to form pipelines, as was defined in Sect. 9.5.2. Over the years, BeepBeep has been involved in multiple case studies including the detection of bugs in video games [56], the runtime monitoring of security policies in Java programs [12, 33], the tracking of packages in the Physical Internet [11], and the identification of electrical appliances in a smart home [25]. BeepBeep's set of processors has also been shown to be expressive enough to encompass other stream languages [26], including LOLA [17], Quantified Event Automata (QEA) [8], and a first-order extension of linear temporal logic [29].

For the purpose of this work, each `Processor` instance provided by the core library has been retrofitted with an additional interface called `Stateful`. This interface defines a single method called `getState`, which returns an arbitrary Java `Object`. The only requirement is that this object must be properly comparable using Java's `equals` method, in order to reliably determine when a processor actually changes its internal state upon ingesting an input event. As we have seen, this condition is essential to optimize the pattern detection procedure we introduced in this chapter and in particular to restrict its output only to the *progressive* subsequences of pattern match instances.

Although not studied in this work, a "cheap" way of obtaining a function returning the internal state of a processor could simply amount to serializing the object and pass the resulting data structure (such as an XML or JSON string) into a hash function. By construction, the resulting value satisfies the conditions of a state function expressed in Sect. 9.4.3, although it may not guarantee optimality. Since, in many cases, the relevant internal state of a processor (for the purpose of pattern recognition) is a subset of the whole internal state of the Java object, we elected for a simpler methodology where a well-chosen object or data structure has been purposefully coded for each processor instance.

In addition to these modifications to the core library, an extension of the system (a plug-in which is called a *palette* in BeepBeep's terminology) providing additional processors has been developed, specifically for the task of detecting patterns in a stream of events. This palette provides a processor called `DetectPattern`, which is a faithful transcription of Algorithm 2. What is more, all the monitors discussed in Sect. 9.5.1 and illustrated in Fig. 9.7 are also defined, along with their corresponding state function.

In order to calculate the progressing subsequences of each processor, the palette takes advantage of an existing mechanism built within BeepBeep and called the *event tracker*. As described in earlier work [24], the task of this object is to record in memory the associations between input events and output events that any processor may want to register during its execution, thereby mirroring the purpose of function $\rho_\pi$ formally defined in Sect. 9.5.2. For these "lineage" capabilities to be active, each processor instance must be associated with an instance of the `EventTracker` class. Since each processor instance in BeepBeep is given a numerical identifier that is unique across a given program, the associations for each processor of a chain can be recorded and distinguished.

Figure 9.9 shows a concrete example of Java code building a BeepBeep pipeline using the processors described in this chapter. Lines 1–7 create the processor

```
 1  CountDecimate d = new CountDecimate(2);
 2  Fork f = new Fork(3);
 3  SomeEventually a = new SomeEventually(new ApplyFunction(new Equals("a")));
 4  SomeEventually b = new SomeEventually(new ApplyFunction(new Equals("b")));
 5  SomeEventually c = new SomeEventually(new ApplyFunction(new Equals("c")));
 6  Sequence s1 = new Sequence();
 7  Sequence s2 = new Sequence();
 8  EventTracker t = new IndexEventTracker();
 9  Connector con = new Connector(t);
10  con.connect(d, 0, f, 0).connect(f, 0, a, 0).connect(f, 1, b, 0)
11    .connect(f, 2, c, 0).connect(a, 0, s1, 0).connect(b, 0, s1, 1)
12    .connect(s1, 0, s2, 0).connect(c, 0, s2, 1);
```

**Fig. 9.9** A Java code snippet creating the pipeline of Fig. 9.8

```java
1  FindPattern fp = new FindPattern(g);
2  Connector.connect(fp, new Print());
3  for (char e : "AabcAaBabbCa".toCharArray()) {
4    fp.getPushableInput().push(e);
5  }
```

**Fig. 9.10** A Java code snippet finding instances of the pattern detected by the pipeline of Figure 9.9 on the input trace $AabcAaBabbCa$

instances, respectively, corresponding to boxes 1–7 in Fig. 9.8. Then, line 8 creates an instance of event tracker, and line 9 creates an instance of the `Connector` object used to link input and output pipes. Finally, method `connect` of this object is repeatedly called to create the appropriate connections between the processors of the pipeline. The connector takes care of both associating each processor with the event tracker and also of registering the connection between these processors into the event tracker itself.

Once this pipeline has been created, it can be encapsulated into a `Group-Processor` `g` and then be passed as a monitor to the `FindPattern` processor, as is shown in Fig. 9.10. For the purpose of this example, the processor is connected to an instance of `Print`, which, as its name implies, prints to the standard output all the events that are fed to it. Lines 3–5 simply iterate over all characters of the input trace (turning them into individual events) and push them into the `FindPattern` processor. The expected output of this piece of code is:

```
{(1,1), (1,7), (1,11)}
```

which corresponds precisely to the positions in the input stream highlighted in Fig. 9.8 and which constitute the progressing subsequence of the match detected by the pipeline.

As one can see, the software implementation very closely follows the theoretical definitions introduced in this chapter. The definition of the pipeline, the detection of the patterns, and the extraction of a progressive subsequence for a given match exactly mirror the expected results from the formalization.

### 9.6.2 Empirical Analysis

Equipped with this implementation, it is now possible to evaluate the potential for reduction in the size of the witnesses by running the `FindPattern` processor on a set of monitors and input streams. The patterns we consider are abstract, but each has a global shape that corresponds to a type of real-world type of attack.

The first pattern is called *Linear sequence*: it looks for a sequence of successive symbols in an input stream, with each symbol separated from the next by an arbitrary number of events. This is a generalization of the pattern detected by the monitor of Fig. 9.8, where the number of symbols in the sequence is configurable

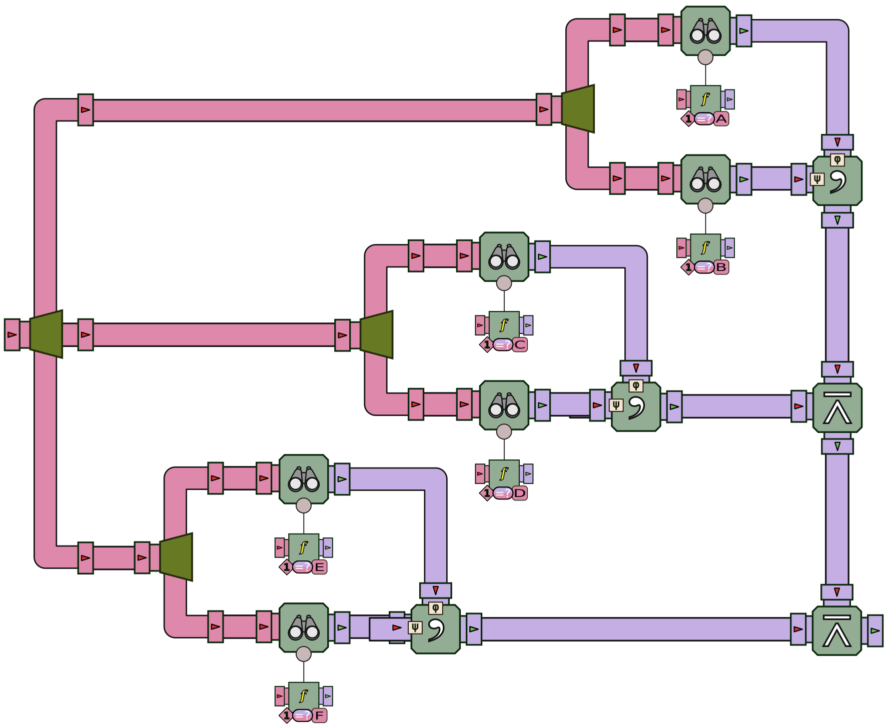


**Fig. 9.11** A monitor identifying the occurrence of three linear patterns ($A \rightarrow B$, $C \rightarrow D$, $E \rightarrow F$), which can be arbitrarily interleaved

by a parameter $n$. This pattern can be used to for any attack that can be detected by looking for a succession of fixed actions, each marking a progression in the unfolding of the attack. An example is the `ptrace` exploit described by Olivain and Goubault-Larrecq [45].

The second pattern is called *Combined*: it is parameterized by $n$ other patterns $\pi_1, \ldots, \pi_n$. It declares a match as soon as all these patterns have been detected in the stream, irrespective of the order in which they occur or the possible interleaving of their relevant events. An illustration of a possible monitor for this pattern is shown in Fig. 9.11. This particular case is parameterized with three instances of the linear sequence pattern introduced above: the first instance detects an $A$ followed by a $B$, while the other two, respectively, detect the sequences $C \rightarrow D$ and $E \rightarrow F$. A possible input stream matching this pattern would be $aCbABbEaAFD$. As its name implies, this pattern is appropriate for attacks that require multiple independent conditions to be present for it to succeed, with each condition itself being detectable by its own pattern in the stream of events; an example is the attack pattern of the GandCrab ransomware [55], which consists of multiple steps, each of them requiring the parallel fulfillment of two or more sequences.

The third pattern is called *Incomplete*. It is parameterized by a sequential pattern that can unfold concurrently on multiple slices of a given stream. A pattern for a given slice is said to be “incomplete” if the first event that defines it has been observed, but the last event concluding the occurrence of the pattern has not yet been received. An alarm is raised when the number of incomplete instances of the pattern at a given moment exceeds some threshold number $k$. A possible incident following this pattern is the classical SYN flooding attack [16]; in this case, an attacker opens a large number of TCP connections by issuing the SYN segment, but does not confirm the start of the connection by issuing the expected ACK segment later on. A large number of incomplete instances of the pattern SYN$\rightarrow$ACK may thus indicate that such an attack is ongoing.

Figure 9.12 shows an example of a monitor for this pattern. It first slices an incoming stream of tuples according to the value of some attribute $A$ (box #1); on each of these sub-streams, it looks for the presence of a pattern where attribute $B$ of the tuple first contains the value $A$, eventually followed by the value $B$ (part #2 of the pipeline). The output of this part of the pipeline thus produces ? if the pattern is incomplete and $\top$ if the pattern has been completed. The second part of the pipeline (box #3) then turns these two values into the numbers 1 and 0, respectively. The *Slice* processor then calculates the sum of the output value for each slice, which corresponds to the number of incomplete pattern instances at a given moment. It then compares it to the threshold value $k$ (box #4); the monitor of box #5 outputs $\top$ (and thus declares a match) as soon as this value exceeds $k$ at some moment in the trace

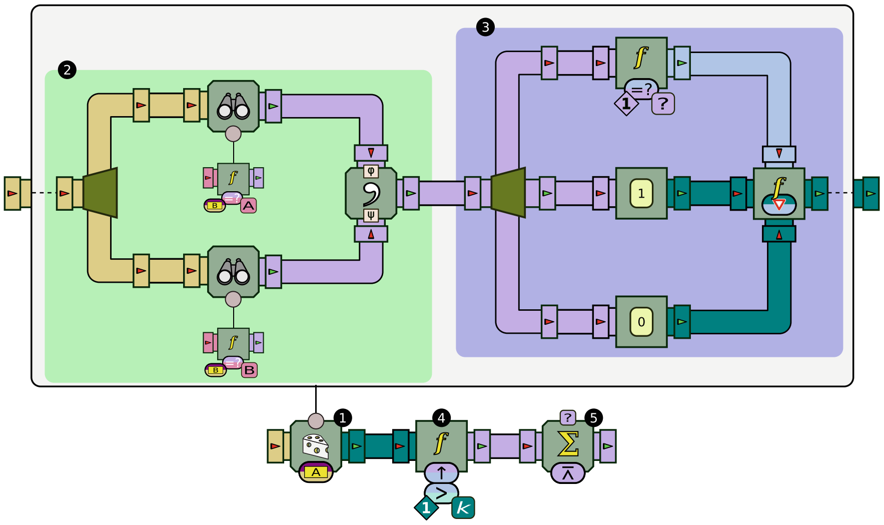


**Fig. 9.12** A monitor triggering an alarm when the number of slices containing an event $A$ not followed by a $B$ exceeds from threshold $k$

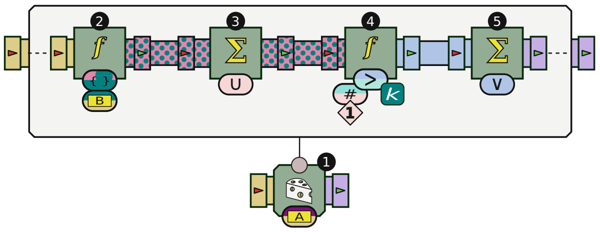


**Fig. 9.13** A monitor identifying a pattern in a sequence of tuples, where a slice determined by the value of attribute $A$ contains at least $k$ distinct values of attribute $B$

The fourth and last pattern is called *Threshold*; in this pattern, values extracted from events in a stream are accumulated into a set; a match is declared when the number of unique values in the set exceeds a parameterizable quantity $k$. An example of an attack following this global pattern is a remote access Trojan [23], which includes a port scanning phase in which multiple TCP connections are initiated by the same peer on various port numbers. In such a case, the port numbers do not necessarily occur following a regular sequence; the pattern matches when enough distinct port numbers are observed, regardless of their order of occurrence.

Figure 9.13 shows an example of a monitor for this pattern. This time, events are assumed to be key-value *tuples* instead of atomic symbols. The pattern is detected by an *Existential slice* monitor, illustrated in box #1. This monitor splits the input stream of events into sub-streams based on the value of attribute $A$ in each tuple. For example, this attribute could be the source address of an incoming IP packet. For each slice, the processing of the succession of processors 2–5 is applied. First, the value of another attribute $B$ in each tuple is extracted and placed into a set (box #2); this attribute could be the TCP port at which a connection is attempted. These values are then aggregated into a set using an instance of the *Cumulate* processor (box #3); the cardinality of this set is repeatedly compared against the threshold quantity $k$ (box #4), producing a stream of Boolean values. These values are then aggregated using the disjunction operator (box #5); the end result is that the pipeline returns $\top$ and keeps returning $\top$ from that point on, when the number of distinct values of attribute $B$ for that slice is higher than $k$. Overall, this pipeline declares a match whenever the number of distinct connection attempts from any individual source IP address becomes too large.

For each of these scenarios, a synthetic event source containing a configurable number of instances of the pattern has been created, using the Synthia data structure generator [48]. A BeepBeep `Source` processor has been implemented, using Synthia's `Knit` object, which can interleave sequences of events generated from multiple instances of the `Picker` interface. Each time a new event it to be generated, `Knit` throws a biased dice to determine which picker instance is chosen to produce the next event. In the present case, two picker instances are configured: the first generates randomly selected events that are not present in the pattern to be discovered, while the second generates one possible next event that makes

**Table 9.1** Number of pattern matches for each simplification strategy, with $\alpha = 0.9$

| Pattern | First step | Direct | Distinct states | Progressing |
|---|---|---|---|---|
| Threshold | 52 | 190 | 45 | 45 |
| Linear sequence | 22 | 247 | 22 | 22 |
| Combined patterns | 7 | 125 | 3 | 3 |
| Incomplete | 223 | 223 | 33 | 33 |

the pattern's monitor move to a new state. A configurable parameter, $\alpha \in [0, 1]$ determines the probability that the "regular" picker is chosen over the "pattern" picker. A low value of $\alpha$ means that events from the pattern are chosen more often, resulting in denser matches, while a high value of $\alpha$ causes pattern instances to be spread across a larger interval of events in the input stream.

The experiments were implemented using the LabPal testing framework [27], which makes it possible to bundle all the necessary code, libraries, and input data within a single self-contained executable file, such that anyone can download and independently reproduce the experiments. A downloadable lab instance containing all the experiments of this paper can be obtained online from GitHub.[6] All the experiments were run on an Intel CORE i5-7200U 2.5 GHz running Ubuntu 18.04, inside a Java 11 virtual machine with the default settings allocating 1746 MB of memory.

Sequences of 500 events have been generated according to each pattern, and for each of them, the number of matches, number of witness events, and running time have been calculated. Table 9.1 shows the impact on the number of matches. As expected, the *Direct* approach generates the most matches, while the *First step* strategy already reduces the number of such matches. The *Distinct states* approach further reduces the number of matches, while the *Progressing* strategy has no impact on the reduction of the number of pattern matches. These results faithfully mirror the abstract example shown in Fig. 9.5.

Table 9.2 shows the fraction of the input stream that is included in the events identified for each match, for all four reduction strategies. Again, one can observe a sharp reduction by moving from the direct, first step, distinct states and progressing subsequence reductions. While the *Direct* approach includes almost all the input stream for two of the scenarios, at the other end of the spectrum, the *Progressing* strategy incurs a reduction down to 20% or less of the input stream – thereby confirming its potential for identifying the relevant events of a pattern. A notable exception is the *Incomplete* pattern, where the reduction is much smaller than for the remaining patterns. This is explained by the fact that in order to explain the fact that the sequence $A \rightarrow B$ is incomplete, one must show not only that an $A$ has occurred but also that all the subsequent are not $B$s. Therefore, a large amount of the input stream ends up being part of the progressive subsequence. This is a

[6] https://github.com/liflab/pattern-detection-lab

**Table 9.2** Fraction of input stream included in witnesses, with $\alpha = 0.9$

| Pattern | First step | Direct | Distinct states | Progressing |
|---|---|---|---|---|
| Threshold | 0.96015936 | 0.96414346 | 0.96015936 | 0.20717132 |
| Linear sequence | 0.42231077 | 0.9920319 | 0.42231077 | 0.17529881 |
| Combined patterns | 0.70119524 | 0.72908366 | 0.6294821 | 0.17928287 |
| Incomplete | 0.936255 | 0.936255 | 0.84462154 | 0.84462154 |

**Table 9.3** Total running time for the pattern detection algorithm for each reduction strategy, with $\alpha = 0.9$

| Pattern | Progressing | Direct | Distinct states | First step |
|---|---|---|---|---|
| Threshold | 1275 ms | 76 ms | 1012 ms | 1092 ms |
| Linear sequence | 10 ms | 19 ms | 8 ms | 7 ms |
| Incomplete | 37 ms | 475 ms | 120 ms | 766 ms |
| Combined patterns | 209 ms | 1307 ms | 95 ms | 210 ms |

common trait shared by patterns that are negative in essence (i.e. that raise an alarm depending on the absence of an event or sequence of events).

Finally, Table 9.3 shows the total running time for the pattern detection algorithm for each reduction strategy. One can observe that running time involves a form of cost-benefit equilibrium. The *Direct* approach incurs an increased number of monitor instances and thus additional computational load resulting in longer running times for the combined pattern scenario. In counterpart, simplification strategies require the evaluation of each monitors' state at each input event, which may end up incurring a non-negligible cost; yet, one can see that this additional effort is offset by the fact that monitor instances are regularly discarded, resulting in an overall quicker running time than the direct approach. Finally, the calculation of the progressing subsequence involves additional effort, which explains why the running time for this last layer of simplification results in longer running times than the *Distinct states* strategy.

Overall, these results are consistent with the expected behavior predicted by theoretical reasoning over the description of Algorithm 2. They show that the use of the progressing subsequence reduction strategy, even though it produces fewer matches and fewer witness events extracted from the input stream, incurs a reasonable overhead due to the fact that superfluous monitor instances are not uselessly updated upon each new event.

## 9.7 Discussion and Conclusion

In this chapter, we have shown how the task of detecting misbehavior or malicious actions over an information system can be reduced to the specification of abstract *monitors* whose simple task is to determine, given a stream of events produced by the observation of the system, whether a given pattern occurs or not. These monitors are simply special cases of a more general function called a *processor*, which consumes input streams in order to produce output streams. The definition of a monitor has been purposefully left as generic as possible, and in particular does not impose any particular notation or formalism to express the patterns of interest.

From these monitors, an algorithm to detect all matches of a given pattern has been proposed. Then, the notion of processor *state* was introduced, which, again, is defined only based on the relationship between possible inputs and possible outputs, without the need to refer to any particular inner implementation detail of the processor. Thus, a state function can be inferred for any processor given its definition, and is not restricted to such processors that are defined as an explicit state machine.

This notion of state has then been leveraged to implement several simplification and optimization layers on the basic monitor-based pattern detection algorithm. More importantly, processor state can be used not only to reduce the number of (mostly superfluous) matches discovered by monitor instances but also to reduce the number of events that are singled out in each match as material "witnesses" of the occurrence of this match – a concept called the *progressing subsequence*. Experimental evaluation on sample patterns inspired from examples of real-world attacks confirm the intuition that these simplification strategies indeed reduce the amount of data extracted from a log required to provide evidence of the occurrence of a particular pattern.

The ideas introduced in this chapter can be extended in several ways: First, the definition of progressive subsequence shares similarities with the definition of progressing subsequence of a processor, as described in Sect. 9.4.3. It presents the interesting property that for processors expressed as Moore (i.e., finite-state) machines, the progressing subsequence of an input trace of events exactly coincides with the definition of what is called an *explanation* in [24]. However, while an explanation is an *ad hoc* definition that was only valid for Moore machines, it is here generalized to arbitrary processors. This links back to a theoretical notion called *explainability*, which was defined by Hallé and Tremblay [28] and which can be paraphrased as follows:

> Let $f : X \rightarrow Y$ be a function, $x \in X$ be an input of $f$, $p_x$ be a part of $x$, and $p_{f(x)}$ be a part of $f(x)$. Part $p_x$ is said to "explain" the output of $f$ if there exists an input $x' \in X$ that differs only on $p_x$ and such that $f(x')$ differs on $p_{f(x)}$.

In addition, the notion of processor state could be used to generate "early warnings" of the occurrence of a pattern. Instead of alerting a user when a monitor declares the occurrence of a complete attack pattern, one could create a derived version of this monitor that declares a match whenever the pattern is almost fulfilled – for example, when there exists an extension of the current input stream by a single event that would cause the original monitor to declare a match.

## References

1. Snort: Network intrusion detection and prevention. https://www.snort.org. Accessed 28 Sept 2022
2. The Zeek network security monitor. https://zeek.org. Accessed 28 Sept 2022
3. J.R. Abrial, *The B-Book: Assigning Programs to Meanings* (Cambridge University Press, 2005)
4. U. Adhikari, T.H. Morris, S. Pan, Applying non-nested generalized exemplars classification for cyber-power event and intrusion detection. IEEE Trans. Smart Grid **9**(5), 3928–3941 (2018). https://doi.org/10.1109/TSG.2016.2642787
5. A. Ahmed, A. Lisitsa, C. Dixon, A misuse-based network intrusion detection system using temporal logic and stream processing, in *5th International Conference on Network and System Security, NSS 2011*, Milan, Italy, 6–8 Sept 2011, ed. by P. Samarati, S. Foresti, J. Hu, G. Livraga (IEEE, 2011), pp. 1–8. https://doi.org/10.1109/ICNSS.2011.6059953
6. M.A. Albahar, Recurrent neural network model based on a new regularization technique for real-time intrusion detection in SDN environments. Secur. Commun. Netw. (2019). https://www.scopus.com/inward/record.uri?eid=2-s2.0-85076009173&doi=10.1155%2f2019%2f8939041&partnerID=40&md5=7a20449e6b871b80dedcded928a20e01.
7. G.E. Andrews, *Number Theory* (Dover, 1994)
8. H. Barringer, Y. Falcone, K. Havelund, G. Reger, D.E. Rydeheard, Quantified event automata: towards expressive and efficient runtime monitors, in *FM*, ed. by D. Giannakopoulou, D. Méry. Lecture Notes in Computer Science, vol. 7436 (Springer, 2012), pp. 68–84
9. E. Bartocci, Y. Falcone, A. Francalanza, G. Reger, Introduction to runtime verification, in *Lectures on Runtime Verification – Introductory and Advanced Topics*, ed. by E. Bartocci, Y. Falcone. Lecture Notes in Computer Science, vol. 10457 (Springer, 2018), pp. 1–33. https://doi.org/10.1007/978-3-319-75632-5_1
10. A. Bédard, S. Hallé, Model checking of stream processing pipelines, in *28th International Symposium on Temporal Representation and Reasoning, TIME 2021*, 27–29 Sept 2021, Klagenfurt, Austria, ed. by C. Combi, J. Eder, M. Reynolds. LIPIcs, vol. 206, pp. 5:1–5:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.TIME.2021.5
11. Q. Betti, R. Khoury, S. Hallé, B. Montreuil, Improving hyperconnected logistics with blockchains and smart contracts. IT Prof. **21**(4), 25–32 (2019)
12. M.R. Boussaha, R. Khoury, S. Hallé, Monitoring of security properties using BeepBeep, in *FPS*, ed. by A. Imine, J.M. Fernandez, J. Marion, L. Logrippo, J. García-Alfaro. Lecture Notes in Computer Science, vol. 10723 (Springer, 2017), pp. 160–169
13. K. Bu, M. Xu, X. Liu, J. Luo, S. Zhang, M. Weng, Deterministic detection of cloning attacks for anonymous RFID systems. IEEE Trans. Ind. Inf. **11**(6), 1255–1266 (2015). https://doi.org/10.1109/TII.2015.2482921
14. E. Börger, *Abstract State Machines: A Method for High-Level System Design and Analysis* (Springer, 2003)
15. H. Chen, Y. Fu, Z. Yan, Survey on big data analysis algorithms for network security measurement, in *Network and System Security – 11th International Conference, NSS 2017*, Helsinki, Finland, 21–23 Aug 2017, Proceedings, ed. by Z. Yan, R. Molva, W. Mazurczyk, R. Kantola. Lecture Notes in Computer Science, vol. 10394 (Springer, 2017), pp. 128–142. https://doi.org/10.1007/978-3-319-64701-2_10
16. Computer Emergency Response Team: TCP SYN flooding and IP spoofing attacks. Tech. Rep. CERT Advisory CA-1996-21, Cybersecurity & Infrastructure Security Agency (1996)
17. B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H.B. Sipma, S. Mehrotra, Z. Manna, LOLA: runtime monitoring of synchronous systems, in *12th International Symposium on Temporal Representation and Reasoning (TIME 2005)*, 23–25 June 2005, Burlington, Vermont, USA (IEEE Computer Society, 2005), pp. 166–174

18. M. Devarajan, L. Ravi, S. Vairavasundaram, V. Varadharajan, A.K. Sangaiah, Hybrid reasoning-based privacy-aware disease prediction support system. Comput. Electr. Eng. **73**, 114–127 (2019). https://doi.org/10.1016/j.compeleceng.2018.11.009
19. V.L. Do, L. Fillatre, I. Nikiforov, P. Willett, Feature article: security of SCADA systems against cyber-physical attacks. IEEE Aerosp. Electron. Syst. Mag. **32**(5), 28–45 (2017)
20. P. Faymonville, B. Finkbeiner, S. Schirmer, H. Torfah, A stream-based specification language for network monitoring, in *Runtime Verification – 16th International Conference, RV 2016*, Madrid, Spain, 23–30 Sept 2016, Proceedings, Y. Falcone, C. Sánchez. Lecture Notes in Computer Science, vol. 10012 (Springer, 2016), pp. 152–168. https://doi.org/10.1007/978-3-319-46982-9_10
21. C.J. Fung, Q. Zhu, FACID: a trust-based collaborative decision framework for intrusion detection networks. Ad Hoc Netw. **53**, 17–31 (2016). https://doi.org/10.1016/j.adhoc.2016.08.014
22. J. Goubault-Larrecq, J. Olivain, A smell of Orchids, in *Runtime Verification, 8th International Workshop, RV 2008*, Budapest, Hungary, 30 March 2008. Selected Papers, ed. by M. Leucker. Lecture Notes in Computer Science, vol. 5289 (Springer, 2008), pp. 1–20. https://doi.org/10.1007/978-3-540-89247-2_1
23. R.A. Grimes, Danger: Remote access Trojans. Security Administrator (2002). https://technet.microsoft.com/en-us/library/dd632947.aspx. Accessed 29 Sept 2022
24. S. Hallé, Explainable queries over event logs, in *24th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2020*, Eindhoven, The Netherlands, 5–8 Oct 2020 (IEEE, 2020), pp. 171–180. https://doi.org/10.1109/EDOC49727.2020.00029
25. S. Hallé, S. Gaboury, B. Bouchard, Activity recognition through complex event processing: first findings, in *Artificial Intelligence Applied to Assistive Technologies and Smart Environments, Papers from the 2016 AAAI Workshop*, Phoenix, Arizona, USA, 12 Feb 2016, ed. by B. Bouchard, S. Giroux, A. Bouzouane, S. Gaboury. AAAI Workshops, vol. WS-16-01 (AAAI Press, 2016)
26. S. Hallé, R. Khoury, Writing domain-specific languages for BeepBeep. In: C. Colombo, Leucker, M. (eds.) RV. Lecture Notes in Computer Science, vol. 11237, pp. 447–457. Springer (2018)
27. S. Hallé, R. Khoury, M. Awesso, Streamlining the inclusion of computer experiments in a research paper. Computer **51**(11), 78–89 (2018)
28. S. Hallé, H. Tremblay, Foundations of fine-grained explainability, in *Computer Aided Verification – 33rd International Conference, CAV 2021, Virtual Event*, July 20–23, 2021, Proceedings, Part II, ed. by A. Silva, K.R.M. Leino. Lecture Notes in Computer Science, vol. 12760 (Springer, 2021), pp. 500–523. https://doi.org/10.1007/978-3-030-81688-9_24
29. S. Hallé, R. Villemaire, Runtime enforcement of web service message contracts with data. IEEE Trans. Serv. Comput. **5**(2), 192–206 (2012)
30. S. Hallé, Event Stream Processing with BeepBeep 3: Log Crunching and Analysis Made Easy. Presses de l'Université du Québec (2018)
31. S. Iqbal, M.L.M. Kiah, B. Dhaghighi, M. Hussain, S. Khan, M.K. Khan, K.R. Choo, On cloud security attacks: a taxonomy and intrusion detection and prevention as a service. J. Netw. Comput. Appl. **74**, 98–120 (2016). https://doi.org/10.1016/j.jnca.2016.08.016
32. A. Kassem, Y. Falcone, Detecting fault injection attacks with runtime verification, in *Proceedings of the 3rd ACM Workshop on Software Protection, SPRO@CCS 2019*, ed. by P. Falcarin, M. Zunke, London, Uk, 15 Nov 2019 (ACM, 2019), pp. 65–76. https://doi.org/10.1145/3338503.3357724
33. R. Khoury, S. Hallé, O. Waldmann, Execution trace analysis using LTL-FOˆ+, in *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications – 7th International Symposium, ISoLA 2016*, Imperial, Corfu, Greece, 10–14 Oct 2016, Proceedings, Part II, ed. by T. Margaria, B. Steffen. Lecture Notes in Computer Science, vol. 9953 (2016), pp. 356–362. https://doi.org/10.1007/978-3-319-47169-3_26

34. C. Kolias, G. Kambourakis, A. Stavrou, S. Gritzalis, Intrusion detection in 802.11 networks: empirical evaluation of threats and a public dataset. IEEE Commun. Surv. Tutorials **18**(1), 184–208 (2016). https://doi.org/10.1109/COMST.2015.2402161
35. T.R.B. Kushal, K. Lai, M.S. Illindala, Risk-based mitigation of load curtailment cyber attack using intelligent agents in a shipboard power system. IEEE Trans. Smart Grid **10**(5), 4741–4750 (2019). https://doi.org/10.1109/TSG.2018.2867809
36. D. Kwon, H. Kim, D. An, H. Ju, DDoS attack volume forecasting using a statistical approach, in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, Lisbon, Portugal, 8–12 May 2017 (IEEE, 2017), pp. 1083–1086. https://doi.org/10.23919/INM.2017.7987432
37. W. Li, W. Meng, L. Kwok, H.H. Ip, Enhancing collaborative intrusion detection networks against insider attacks using supervised intrusion sensitivity-based trust management model. J. Netw. Comput. Appl. **77**, 135–145 (2017). https://doi.org/10.1016/j.jnca.2016.09.014
38. G. Liang, J. Zhao, F. Luo, S.R. Weller, Z.Y. Dong, A review of false data injection attacks against modern power systems. IEEE Trans. Smart Grid **8**(4), 1630–1638 (2017). https://doi.org/10.1109/TSG.2015.2495133
39. G. Logeswari, S. Bose, T. Anitha, An intrusion detection system for SDN using machine learning. Intell. Autom. Soft Comput. **35**(1), 867–880 (2023). https://www.scopus.com/inward/record.uri?eid=2-s2.0-85132133653&doi=10.32604%2fiasc.2023.026769&partnerID=40&md5=e0907be624a0048eda2192a876e4808e. Cited by: 0; All Open Access, Hybrid Gold Open Access
40. P. Mishra, E.S. Pilli, V. Varadharajan, U.K. Tupakula, Intrusion detection techniques in cloud environment: a survey. J. Netw. Comput. Appl. **77**, 18–47 (2017). https://doi.org/10.1016/j.jnca.2016.10.015
41. R. Mitchell, I. Chen, Behavior rule specification-based intrusion detection for safety critical medical cyber physical systems. IEEE Trans. Dependable Secur. Comput. 12(1), 16–30 (2015), https://doi.org/10.1109/TDSC.2014.2312327
42. P. Moosbrugger, K.Y. Rozier, J. Schumann, R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. Formal Methods Syst. Des. **51**(1), 31–61 (2017). https://doi.org/10.1007/s10703-017-0275-x
43. P. Naldurg, K. Sen, P. Thati, A temporal logic based framework for intrusion detection, in *Formal Techniques for Networked and Distributed Systems – FORTE 2004, 24th IFIP WG 6.1 International Conference*, Madrid, Spain, 27–30 Sept 2004, Proceedings, ed. by D. de Frutos-Escrig, M. Núñez. Lecture Notes in Computer Science, vol. 3235 (Springer, 2004), pp. 359–376. https://doi.org/10.1007/978-3-540-30232-2_23
44. L. Nishani, M. Biba, Machine learning for intrusion detection in MANET: a state-of-the-art survey. J. Intell. Inf. Syst. **46**(2), 391–407 (2016). https://doi.org/10.1007/s10844-015-0387-y
45. J. Olivain, J. Goubault-Larrecq, The Orchids intrusion detection tool, in *Computer Aided Verification, 17th International Conference, CAV 2005*, Edinburgh, Scotland, UK, 6–10 July 2005, Proceedings, ed. by K. Etessami, S.K. Rajamani. Lecture Notes in Computer Science, vol. 3576 (Springer, 2005), pp. 286–290. https://doi.org/10.1007/11513988_28
46. M.S. Parwez, D.B. Rawat, M. Garuba, Big data analytics for user-activity analysis and user-anomaly detection in mobile wireless network. IEEE Trans. Ind. Inf. **13**(4), 2058–2065 (2017). https://doi.org/10.1109/TII.2017.2650206
47. K. Peng, V.C.M. Leung, Q. Huang, Clustering approach based on mini batch Kmeans for intrusion detection system over big data. IEEE Access **6**, 11897–11906 (2018). https://doi.org/10.1109/ACCESS.2018.2810267
48. M. Plourde, S. Hallé, Synthia: a generic and flexible data structure generator, in *44th 2022 IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022*, Pittsburgh, PA, USA, 22–24 May 2022 (IEEE, 2022), pp. 207–211. https://doi.org/10.1109/ICSE-Companion55297.2022.9793796
49. J. Ren, J. Guo, W. Qian, H. Yuan, X. Hao, H. Jingjing, Building an effective intrusion detection system by using hybrid data optimization based on machine learning algorithms. Secur. Commun. Netw. (2019). https://www.scopus.com/inward/record.

uri?eid=2-s2.0-85068853458&doi=10.1155%2f2019%2f7130868&partnerID=40&md5=f611d318049034805c5f1c83aefaeba7. Cited by: 48; All Open Access, Gold Open Access, Green Open Access

50. K. Rina, S. Nath, N. Marchang, A. Taggu, Can clustering be used to detect intrusion during spectrum sensing in cognitive radio networks? IEEE Syst. J. **12**(1), 938–947 (2018). https://doi.org/10.1109/JSYST.2016.2584098
51. A.S. Sadiq, B.Y. Alkazemi, S. Mirjalili, N. Ahmed, S. Khan, I. Ali, A.K. Pathan, K.Z. Ghafoor, An efficient IDS using hybrid magnetic swarm optimization in wanets. IEEE Access **6**, 29041–29053 (2018). https://doi.org/10.1109/ACCESS.2018.2835166
52. F. Sakiz, S. Sen, A survey of attacks and detection mechanisms on intelligent transportation systems: VANETs and IoV. Ad Hoc Netw. **61**, 33–50 (2017). https://doi.org/10.1016/j.adhoc.2017.03.006
53. J.M. Spivey, *The Z Notation: A Reference Manual* (Prentice Hall, 1989)
54. L.N. Tidjon, M. Frappier, A. Mammar, Intrusion detection systems: a cross-domain overview. IEEE Commun. Surv. Tutorials **21**(4), 3639–3681 (2019). https://doi.org/10.1109/COMST.2019.2922584
55. L.N. Tidjon, M. Frappier, A. Mammar, Intrusion detection using ASTDs, in *Advanced Information Networking and Applications – Proceedings of the 34th International Conference on Advanced Information Networking and Applications, AINA-2020*, Caserta, Italy, 15–17 April, ed. by L. Barolli, F. Amato, F. Moscato, T. Enokido, M. Takizawa. Advances in Intelligent Systems and Computing, vol. 1151 (Springer, 2020), pp. 1397–1411. https://doi.org/10.1007/978-3-030-44041-1_118
56. S. Varvaressos, K. Lavoie, S. Gaboury, S. Hallé, Automated bug finding in video games: a case study for runtime monitoring. Comput. Entertain. **15**(1), 1:1–1:28 (2017)
57. B. Wehbi, E.M. de Oca, M. Bourdellès, Events-based security monitoring using MMT tool, in *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012*, Montreal, QC, Canada, 17–21 April 2012, ed. by G. Antoniol, A. Bertolino, Y. Labiche (IEEE Computer Society, 2012), pp. 860–863. https://doi.org/10.1109/ICST.2012.188
58. A. Woodruff, M. Stonebraker, Supporting fine-grained data lineage in a database visualization environment, in *Proc. ICDE*, 1997, pp. 91–102. https://doi.org/10.1109/ICDE.1997.581742
59. G. Xu, Y. Cao, Y. Ren, X. Li, Z. Feng, Network security situation awareness based on semantic ontology and user-defined rules for internet of things. IEEE Access **5**, 21046–21056 (2017). https://doi.org/10.1109/ACCESS.2017.2734681
60. S.C. Yip, K. Wong, W.P. Hew, M.T. Gan, R.C.W. Phan, et S.-W. Tan, Detection of energy theft and defective smart meters in smart grids using linear regression. Int. J. Electr. Power Energy Syst. **91**, 230–240 (2017)
61. J. Zhang, Z. Chu, L. Sankar, O. Kosut, Can attackers with limited information exploit historical data to mount successful false data injection attacks on power systems? IEEE Trans. Power Syst. **33**(5), 4775–4786 (2018)
62. W. Zhu, M. Deng, Q. Zhou, An intrusion detection algorithm for wireless networks based on ASDL. IEEE CAA J. Autom. Sinica **5**(1), 92–107 (2018). https://doi.org/10.1109/JAS.2017.7510754
63. R. Zuech, T.M. Khoshgoftaar, R. Wald, Intrusion detection and big heterogeneous data: a survey. J. Big Data **2**, 3 (2015), https://doi.org/10.1186/s40537-015-0013-4

# Chapter 10
# Toward Anomaly Detection Using Explainable AI

**Manh-Dung Nguyen, Vinh-Hoa La, Wissam Mallouli, Ana Rosa Cavalli, and Edgardo Montes de Oca**

**Abstract** Anomaly detection in networks is an important aspect of network security, enabling organizations to identify and respond to unusual patterns of activity that may indicate a security threat or performance issue. By identifying and addressing anomalies in real time, organizations can reduce the risk of data breaches and other security incidents and ensure the optimal performance and reliability of their network infrastructure. However, implementing effective anomaly detection in networks with good quality is a significant challenge, requiring careful consideration of several key factors. One of the main challenges of anomaly detection in networks is the sheer volume of data that must be processed and analyzed. Networks generate vast amounts of traffic data, making it difficult to identify patterns and anomalies in real time. To address this challenge, anomaly detection systems must be able to handle large amounts of data and operate at high speeds while also minimizing false positives and false negatives. In this chapter, we present MMT a monitoring framework developed by the Montimage research team to perform anomaly detection. This framework is being extended with explainable AI (XAI) capabilities to better understand the classification done by AI-/ML-based algorithms. The first experimentations are presented in this book chapter using SHAP, LIME, and SHAPASH technologies.

**Keywords** Network monitoring · Anomaly detection · Explainable AI

M.-D. Nguyen (✉) · V.-H. La · W. Mallouli · A. R. Cavalli · E. M. de Oca
Montimage EURL, Paris, France
e-mail: Manh-Dung.Nguyen@montimage.com; Vinh-Hoa.La@montimage.com; Wissam.Mallouli@montimage.com; ana.cavalli@montimage.com; edgardo.montesdeoca@montimage.com

A. Sadovykh et al. (eds.), *CyberSecurity in a DevOps Environment*,
https://doi.org/10.1007/978-3-031-42212-6_10

## 10.1 Introduction

Anomaly detection (AD) techniques can be used to identify a wide range of network anomalies, including network intrusions, malware infections, denial-of-service (DoS) attacks, and other forms of malicious activities [1, 2]. There are several different approaches to anomaly detection in networks, including rule-based methods, statistical methods, and machine learning methods. *Rule-based methods* involve defining a set of rules that describe normal network activity and then flagging any activity that deviates from these rules as anomalous. *Statistical methods* involve using probability distributions and statistical models to identify deviations from normal network activity. *Machine learning* methods involve training algorithms on large datasets of network activity to identify patterns and anomalies.

One of the key challenges of anomaly detection in networks is minimizing false positives, which occur when normal network activities are incorrectly classified as anomalous, and false negatives, obtained when anomalous activities are not detected [3]. To address this challenge, many anomaly detection systems use a combination of multiple techniques, as well as feedback loops and manual review by security analysts.

Other challenges are the need of network activity continuous monitoring and the quality of detecting anomalies. In fact, anomaly detection systems have not only to be able to identify anomalies but also to accurately detect true ones and quickly respond to them, in order to minimize the risk of security incidents or performance issues. The first challenge can be faced through the combination of automated detection methods and human oversight, as well as ongoing analysis and refinement of the detection algorithms. Moreover, AD systems must avoid misclassifications, e.g., normal network activities detected as anomalous. This requires a deep understanding of network behavior and the ability to adapt to changing patterns of activity over time [4].

One approach to anomaly detection is to use *explainable artificial intelligence* (XAI) techniques [5], which are designed to provide transparency and interpretability into the decisions made by the algorithm. This can help network administrators understand why certain network activity is classified as anomalous and can provide insights into potential security threats or performance issues. For example, an XAI-based anomaly detection system might identify a sudden surge in network traffic from a particular IP address as anomalous. By providing explanations of how the algorithm came to that decision, the system can help the network administrator understand that the IP address is engaged in potentially malicious activity, such as a distributed denial-of-service (DDoS) attack.

In this chapter, we rely on *Montimage Monitoring Tool* (MMT) [6], i.e., a set of modules to perform real-time analysis or post-analysis of captured traffic, combined with AI/machine learning (ML) algorithms to classify sessions and detection deviations from learned behaviors. Through this tool, we will perform network monitoring and anomaly detection, also introducing the possibility of using XAI for network traffic classification. The preliminary outcomes of our experimentation will help to extend the AI-based MMT monitoring framework with transparency and interpretability.

The chapter is organized as follows: Section 10.2 will present the MMT architecture and its usage for anomaly detection and network classification. Section 10.3 will propose the usage of XAI for the classification of network traffic. Several algorithms like SHAP and LIME are presented and included in MMT as plugins to existing deep learning (DL) algorithms. Section 10.4 will present the first results demonstrating the interest of using XAI in network traffic classification in general and in anomaly detection in particular.

## 10.2 Network Monitoring Approaches: MMT Monitoring Framework Example

*Network monitoring* is the process of observing and analyzing the performance and security of a computer network [7]. It involves collecting and analyzing data on network activity, such as the amount of data being transmitted, the types of data being transmitted, and the sources and destinations of the data.

*Classification* is an important aspect of network monitoring. It involves identifying network traffic and categorizing it into different types, such as email, web browsing, or file sharing. This can be done through the use of machine learning algorithms, which analyze patterns in the network traffic to identify different types of activity. By classifying network traffic, network administrators can identify potential security threats, such as suspicious or unauthorized activity, and take appropriate action to mitigate the risk. They can also gain insight into network usage, identifying trends and patterns that can help optimize network performance and improve user experience.

Overall, network monitoring and classification are critical components of maintaining a secure and efficient computer network. Through the use of advanced algorithms and analysis techniques, network administrators can gain a deeper understanding of their network and take proactive steps to ensure its continued success.

In the remainder of this section, we present (i) classification techniques, i.e., rule-based classification and AI-based classification, in Sect. 10.2.1 and (ii) the global architecture of Montimage Monitoring Tool and its application for anomaly detection, respectively, in Sects. 10.2.2 and 10.2.3.

### *10.2.1 Classification Techniques*

#### 10.2.1.1 Rule-Based Network Classification

It is a method of categorizing network traffic based on a set of predefined rules, which are typically based on attributes such as the used protocol, source and destination IP addresses, and port numbers. The process of rule-based classification involves (i) examination of data packets as they move through the network and (ii) comparison of their attributes to a set of predefined rules. When a packet matches a rule, it is classified accordingly. For example, a packet that is identified as HTTP traffic (based on the used protocol) with a destination port of 80 (which is typically used for web traffic) might be classified as "web browsing."

Rule-based classification can be effective in identifying certain types of network traffic, such as web browsing, email, or file sharing. However, it can also be limited by its *inflexibility*, since it relies on predefined rules that may not capture all types of network traffic. In addition, rule-based classification can be *vulnerable to evasion techniques* used by attackers to disguise their activities, e.g., using nonstandard ports or encryption.

Despite these limitations, rule-based classification is still a widely used method of network traffic analysis, especially in situations where the network environment is well understood and the types of traffic are relatively stable. It can be an efficient way to identify and filter out unwanted or malicious traffic and provide insights into network usage and performance.

#### 10.2.1.2 AI-Based Network Classification

It involves the use of AI and ML algorithms to identify and categorize network traffic. These algorithms are trained on large datasets of network traffic and use statistical models to classify new data based on their patterns and their features. AI-based network classification can be more *flexible* and *accurate* than rule-based classification, since it can learn from data and adapt to new and evolving types of network traffic. For example, an AI-based classification system might be able to identify previously unknown types of traffic, such as a new type of malware or an emerging application protocol.

There are several different types of AI-based classification techniques, including supervised learning, unsupervised learning, and deep learning. Supervised learning involves training the algorithm on a labeled dataset, where the correct category of each data point is known. Unsupervised learning involves discovering patterns and relationships in unlabeled data, which can be useful for identifying new and previously unknown types of traffic. Deep learning involves training neural networks with multiple layers to learn complex data representations.

AI-based network classification has several advantages over traditional classification methods. It can provide greater accuracy and speed, allowing network administrators to quickly identify and respond to potential security threats. It can also be more scalable, since it can learn from large datasets and adapt to new types of traffic over time.

However, AI-based classification also requires significant computational resources and expertise to develop and maintain. It also raises concerns around privacy and security, since large amounts of sensitive network traffic data are required to train the algorithms. Therefore, it is important to carefully consider the risks and benefits of AI-based network classification before implementing it in a network environment.

### *10.2.2 Global MMT Monitoring Architecture*

The MMT monitoring framework is an open-source monitoring solution developed by Montimage and freely available for the research community on GitHub [6]. Its workflow is presented in Fig. 10.1, and hereinafter, we analyze each MMT component.

#### 10.2.2.1 Feature Extraction

It is the functionality of the module "MMT-Extract" that allows to parse the network traffic, identify sessions, and compute packet and session attributes called features. This module is implemented as a C library that analyzes network traffic

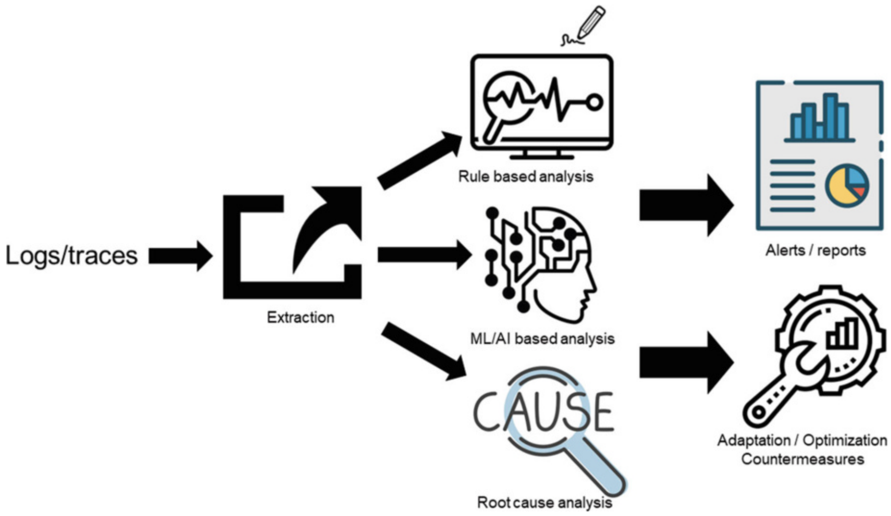


**Fig. 10.1** Monitoring components of MMT

to extract network- and application-based events. Extraction is powered by a plugin architecture that allows adding new protocols or application message formats to parse. In the current development, more than 600 plugins for classical protocols and applications are already implemented.

#### 10.2.2.2 Rule-Based Analysis

"MMT-Security" is a signature-based monitoring solution that allows analyzing network traffic according to a set of properties. These properties contain signatures that formally specify security goals or malicious behaviors related to the monitored system. The MMT-Security property model is inspired by linear temporal logic (LTL) and can be referred to the following two types of properties:

1. Properties that describe the *normal, legitimate behavior* of the application or protocol under analysis. Consequently, the non-respect of the property indicates a potential violation of a safety or security requirement; e.g., all the ports in a computer must be closed unless they are being used by an authorized application.
2. Attacks that describe *malicious behavior* corresponding to an attack model, a vulnerability or misbehavior. In this case, the respect of the property indicates the detection of a potential incident; e.g., a big number of requests in a short period of time could be a DoS attack.

The chosen language of "MMT-Security" properties is XML format, due to its simplicity and straightforward structure verification. A property is a general ordered tree as shown in Fig. 10.2, where the leaf nodes are the atomic events captured in the traces. Each property is composed of a context, in the left branch, and a trigger, in the right branch. Then, a property is valid when the trigger is valid, and the trigger is inspected only if the context is valid.

#### 10.2.2.3 Machine Learning-Based Anomaly Detection

"MMT-AI" allows to perform AI-based analysis of the collected features applying one or several AI/ML algorithms. It is responsible for building a model (which depends on the data and the chosen ML algorithm), as well as utilizing existing ones. We can therefore distinguish its two modes of operating, i.e., training and prediction.

- *Training*: It is designed to create and parameterize the model based on already cleaned and transformed data. This means that it executes the algorithm selected by user and a model is built by using the training data in order to find its weights and biases that would lead to the best results. Loss function, with penalized bad prediction, is used as a metrics of result during training. Depending on the selected algorithm, this step also includes the experimentation of algorithm

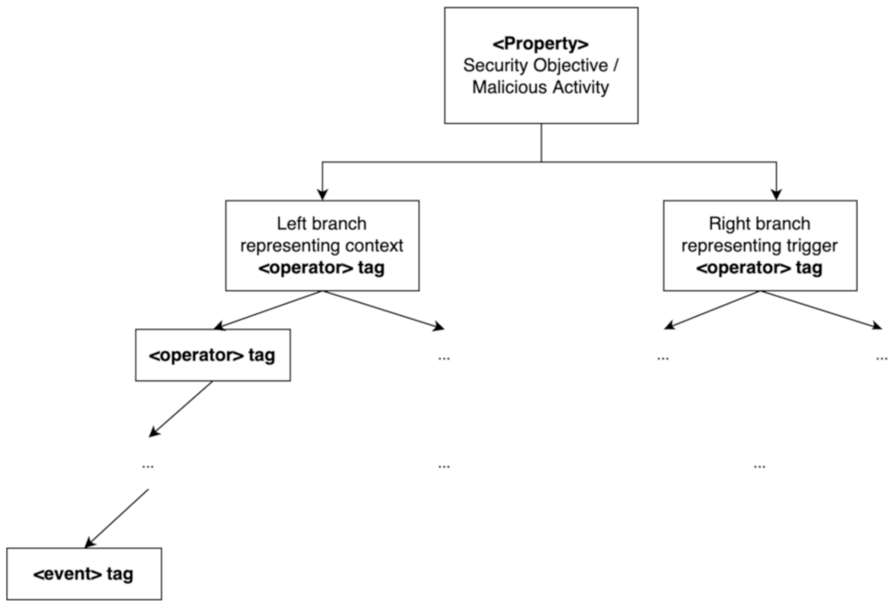

<Property>
Security Objective / Malicious Activity
Left branch representing context <operator> tag
Right branch representing trigger <operator> tag
<operator> tag
<event> tag

**Fig. 10.2** Security property structure in MMT [6]

parameter different values, such as learning rate, activation functions, batch size, and so on. In the proposed system, it is assumed that this step is either done by a user a bit familiar with hyper-tuning or done by utilization of the values directly suggested by the system:

- *Model evaluation*. In order to evaluate the sufficient amount of parameters and the model training, the training needs to be done using a separate dataset from the testing dataset; thus, the model will be tested on the completely new samples. In this case, the evaluation checks whether the model is generalized enough. To investigate the results, the correctness of classification is verified using the following terms: (i) *true positive* (TP) and *true negative* (TN) are samples that are correctly assigned to the normal and anomaly classes, respectively, and (ii) *false positive* (FP) and *false negative* (FN) are samples that are incorrectly assigned to positive or negative classes.

- *Prediction*: It is the activity done after the model is trained (that is to say the algorithm is executed in training mode) and the satisfactory results are obtained. It involves utilizing the model directly on new, unseen data (in a real-life case scenario, these are just production data) and obtaining the results, such as probabilities, classifications, etc. Importantly, the accuracy of the prediction results can also be used in order to further hyper-tune the model.

As the system aims to simplify the prototyping and utilization of AI/ML algorithms for practical applications, it is assumed that the user may want to create multiple different models. Therefore, instead of selecting one model's predictions from one particular model, it can be beneficial to combine the results of different models together. Thus, this final (and optional) step of the ML module consists of the ensemble part that is capable of joining the results together.

#### 10.2.2.4 Root Cause Analysis

"MMT-RCA" relies on machine learning algorithms to identify the most probable cause(s) of detected anomalies based on the knowledge of similar observed ones. It enables the systematization of the experience in dealing with incidents to build a historical database and verify whether a newly detected incident is similar enough to an observed one with known causes. Thanks to MMT-RCA's suggestions, remediation actions could be timely and wisely taken to prevent or mitigate the damage of the recurrence of problems.

### *10.2.3 Application of MMT for Anomaly Detection*

MMT-AI has been used in several projects, e.g., for differentiating bots and human activities in the net [8], for anomaly detection in industrial systems (e.g., load position system of ABB) [9]. In the following, we present a classical usage of MMT-AI on an open-source database CSE-CIC-IDS2018 provided by the Canadian Institute for Cybersecurity [10].

#### 10.2.3.1 Settings

*Stacked autoencoders* (SAE) [11] and *convolutional neural network* (CNN) [12] are used to train and classify the network traffic with the Canadian dataset. More in detail, SAE are multiple encoders stacked on top of one another. The number of neurons in each decoder and encoder is the same. They aim at dimensionality reduction, i.e., filtering the essential features from the data. Then, CNNs are used, and they are a specialized type of artificial neural networks that use, in at least one of their layers, a mathematical operation called convolution in place of general matrix multiplication. It consists of an input layer; hidden layers, which perform convolutions; and an output layer. The general implemented architecture can be seen in Fig. 10.3. The advantage of this architecture is its flexibility, as both modules can be easily added to the structure of the global system and integrated in the final solution.

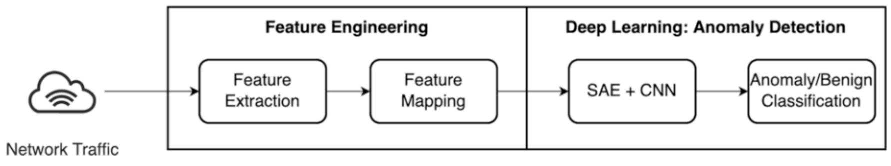


**Fig. 10.3** The AI-based anomaly detection architecture

*Feature Extraction* The feature extractor module is used in both generating the training/testing datasets and on the input file for prediction.

*Scaling* Data scaling is the treatment data process in order to obtain standard format data; thus, the training is improved, accurate, and faster. Indeed, a model with large weight values is often unstable, which means that it may give poor performance during learning and have a high sensitivity to the input values, which leads to a higher generalization error. Column normalization involves bringing the column values to a common scale, which is usually done for columns with varying ranges.

***MinMaxScaler*** from scikit-learn library [13] transforms features by scaling each feature to a given range. This estimator individually scales and translates each feature; in this way, values are limited to a given range in the training set, e.g., between zero and one. The transformation is given by the following equations:

$$X_{std} = \frac{X - X_{min}}{X_{max} - X_{min}} \tag{10.1}$$

$$X_{scaled} = X_{std} \times (X_{max} - X_{min}) + X_{min} \tag{10.2}$$

where $(X_{min}, X_{max})$ represents the desired range of scaled data, e.g., (0, 1).

*Training* In the learning phase, the model is fed with the so-called training dataset, and the model is tested in order to obtain the best performance and highest accuracy of the final classification. The input files pass through the feature extractor module which runs the MMT-Extract. Then, it creates a training and testing .csv files with balanced 0/1 classes. More in detail, the dataset is divided in this way: 70% for training and 30% for testing.

At this level, there is the possibility that we must do multiple experiments with the use of different parameters of the models, e.g., CNN or SAE. Thus, additional model adaptation toward specific conditions or changes is recommended in order to obtain higher performance and accuracy of the model that will be saved for prediction purpose later. The model's structure is hybrid, composed of two auto-encoders and one-dimensional CNN as shown in Fig. 10.4.

*Evaluation* We applied the learned model to the 30% of remaining datasets. The classification is done and the results are presented in the next subsection.

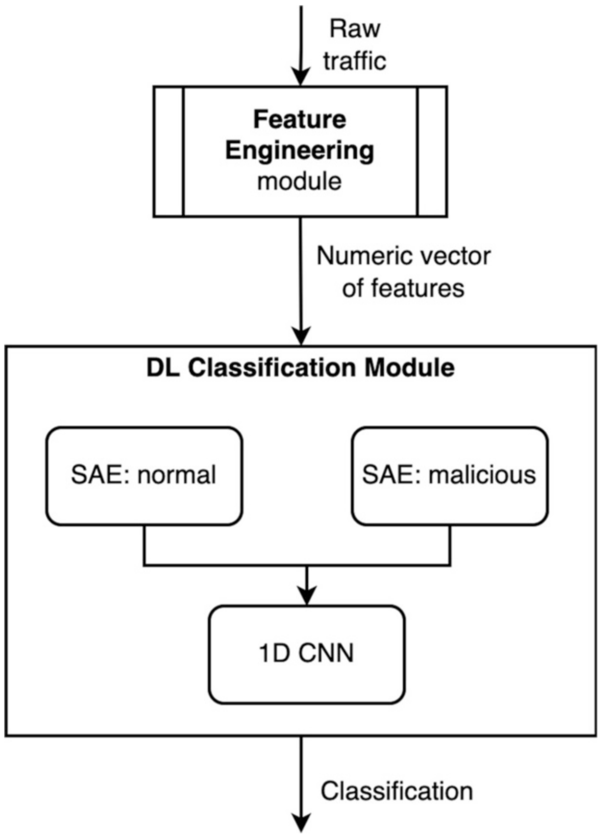


**Fig. 10.4** Overview of the deep learning modules

#### 10.2.3.2 Results and Interpretation

Using the default parameters, the training on these datasets gives the following results, as shown in Tables 10.1 and 10.2.

The obtained results are quite impressive and are more than 99% for the precision, recall, and F1-score, as shown in Tables 10.3 and 10.4. However, better results are still possible by refining parameters.

The same methodology has been applied to real traffic data collected in the Montimage internal network (private network). The train and test datasets present 15000 samples together, and as already said, the dataset has been split into 70% for training and 30% for testing. The data are shuffled in order to give different patterns of the presence of 0/1 (normal/malicious) samples in the files. After the training process on these data, we obtained the following results, as shown in Tables 10.5 and 10.6.

**Table 10.1** Confusion matrix 1

| | 0 | 1 |
|---|---|---|
| 0 | 6779 | 14 |
| 1 | 3 | 6790 |

**Table 10.2** Metrics of model using default parameters

| | Precision | Recall | F1-score | Support |
|---|---|---|---|---|
| 0 (Normal traffic) | 0.999558 | 0.997939 | 0.998748 | 6793 |
| 1 (Malware traffic) | 0.997942 | 0.999558 | 0.998750 | 6793 |
| Accuracy | 0.998749 | 0.998749 | 0.998749 | 0.998749 |
| Macro average | 0.998750 | 0.998749 | 0.998749 | 13586 |
| Weighted average | 0.998750 | 0.998749 | 0.998749 | 13586 |

**Table 10.3** Confusion matrix 2

| | 0 | 1 |
|---|---|---|
| 0 | 6778 | 5 |
| 1 | 3 | 6790 |

**Table 10.4** Metrics of model using advanced parameters

| | Precision | Recall | F1-score | Support |
|---|---|---|---|---|
| 0 (Normal traffic) | 0.999558 | 0.999264 | 0.999411 | 6793 |
| 1 (Malware traffic) | 0.999264 | 0.999558 | 0.999411 | 6793 |
| Accuracy | 0.999411 | 0.999411 | 0.999411 | 0.999411 |
| Macro average | 0.999411 | 0.999411 | 0.999411 | 13586 |
| Weighted average | 0.999411 | 0.999411 | 0.999411 | 13586 |

**Table 10.5** Confusion matrix 3

| | 0 | 1 |
|---|---|---|
| 0 | 7500 | 0 |
| 1 | 87 | 7413 |

**Table 10.6** Metrics of model using real network traffic

| | Precision | Recall | F1-score | Support |
|---|---|---|---|---|
| 0 (normal traffic) | 0.999558 | 0.999264 | 0.999411 | 6793 |
| 1 (malware traffic) | 0.999264 | 0.999558 | 0.999411 | 6793 |
| Accuracy | 0.999411 | 0.999411 | 0.999411 | 0.999411 |
| Macro average | 0.999411 | 0.999411 | 0.999411 | 13586 |
| Weighted average | 0.999411 | 0.999411 | 0.999411 | 13586 |

The results were good enough to affirm that the obtained model is efficient. Thus, we further investigated the classification phase in order to check the accuracy of the prediction of the model. We used a portion of the raw normal traffic data that we used for training. In this way, we know that the model is correctly functioning when we see zeros in the predicted malware value. However, the results were not compatible with the expected values, and the predictions are thus not correct. The model predicts that there were attacks in the known normal traffic. Therefore, further investigation and testing need to be done.

*Discussion* The manual investigation shows that the classification using AI-based anomaly detection provides in some cases false positives (e.g., 0.1% for malware prediction using default parameters) which are difficult to interpret mainly with theoretical metrics that are more than 99% (precision, recall, F1-score, etc.). The need to have more transparency is needed in such context to better interpret the results and understand why we have such decisions. That's why we will use XAI to have a better insight on network traffic classification using explainable AI. These results are still preliminary.

## 10.3 Interpreting ML Models for User Network Activity Classification

### *10.3.1 Motivation*

#### 10.3.1.1 Context

Network traffic classification becomes more and more challenging due to the growth in network traffic. As there are new applications with different characteristics and network requirements, it is crucial to identify the requirements to provide the appropriate resource to each application. In the literature, several approaches have been proposed for network traffic classification based on the well-known ports (e.g., TCP or UDP port numbers) and on deep packet inspection (DPI) technique [14]. However, port-based classification technique is ineffective because mapping between ports and applications using dynamic ports is not well defined. Moreover, the growing popularity of *encrypted traffic* HTTPS and virtual private networks (VPN) increases user security and privacy but also becomes a big challenge for traditional traffic analysis, making DPI-based service classification unfeasible. Therefore, it raises the need for advanced analysis techniques based on other criteria, such as behavior analysis. With the introduction of network encryption techniques, such as the TLS protocol, the accuracy and efficiency of conventional Network Intrusion Detection Systems (NIDS) that were using rule- and signature-based monitoring detection methods are greatly reduced. Consequently, in the last decade, research efforts have moved toward new analysis methods based on AI techniques for network traffic classification. Indeed, various AI algorithms have been used in the literature, such as supervised [14, 15], unsupervised [16], and hybrid machine learning approaches [17–19].

Nowadays, apart from accuracy and performance, new requirements concerning trustworthy, transparency, unbiasedness, privacy, and robustness also need to be taken into account in the development of AI-based systems. Nonetheless, existing AI methods, especially complex ones like deep neural networks, are seen as black boxes and thus have a common limitation of *lacking explainability*. Indeed, the classification results of existing work do not provide the users with any information of how the dataset, input features, or selected models contribute to the predicted classification. In this context, user network activity classifiers, as well as other traffic analysis applications, must be improved and optimized not only in terms of performance but also for other properties listed above. Recently, *explainable artificial intelligence (XAI)* has become a hot research topic in the AI community [20]. It provides a rationale that allows users to understand why an AI-based system has produced a given output and increases trust of end users. Different approaches [21, 22] are proposed to providing and improving the understanding, in the global and local manner, of what the models have learned and how the models make individual predictions.

#### 10.3.1.2 Proposal

Our work aims at characterizing and classifying user network activities using machine learning techniques. We use popular supervised techniques, such as random forest, neural networks, XGBoost,[1] and LightGBM,[2] and unsupervised techniques, such as K-means, for classification. Furthermore, we want not only to understand why our application produces promising results but also why it makes some wrong predictions in some cases to further improve the performance. To achieve this goal, we add an extra explainability layer on top of our AI-based classification system by applying different popular XAI methods, such as SHAP and LIME. The full dataset and the code of the AI-based system will be published along with this paper at [23].

### *10.3.2 Classification of User Network Activities*

#### 10.3.2.1 Overview

Our classification system takes as input network traffic data with IP and TCP/UDP header fields. Figure 10.5 illustrates an overview of the workflow of our AI-based classification consisting of four main phases: *dataset generation*, *dataset preprocessing*, *feature extraction*, and *classification*. The dataset generation process is indeed important for training and testing our application. The dataset preprocessing phase is required to describe and transform the input network traffic data into a set of features suitable for the classification task. Then, different classification models are executed using the feature selection output to predict the user activity in one of

[1] https://xgboost.readthedocs.io

[2] https://lightgbm.readthedocs.io

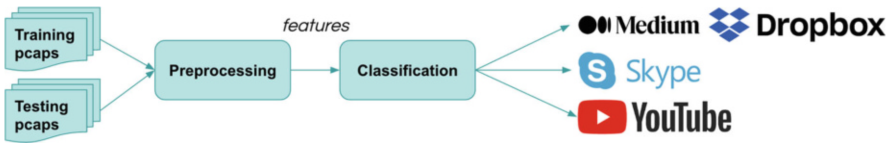


**Fig. 10.5** Overview of AI-based classification for user activities

the three groups: Web, interactive, and video. Below, we describe the four phases in more details:

### 10.3.2.2 Types of Activities

We choose the most common user activities on the Internet covering behaviors exhibited by the network traffic from different applications. The set of three classes are as follows:

1. *Web browsing* activity includes the network traffic generated when users search or view different web pages, including downloading of multimedia content such as text, images, or advertising video. This activity can also include traffic of applications that transfer big volume data over the network.
2. *Interactive* activity contains network traffic of applications that execute real-time interactions, for example, chatting application like Discord and Messenger or remotely editing files on Google Docs.
3. *Video* activity contains network traffic of applications consuming video in streaming mode, for example, watching online movies or YouTube videos.

### 10.3.2.3 Dataset Generation

Figure 10.6 depicts the overview of our methodology used to capture network traffic to generate the dataset for training and evaluating the classification system. First, we capture the traffic using Wireshark [24] to produce pcap files whose size is 2.15 Gb when a user performs some normal activities on a single host. Concretely, for each class, we perform the following activities: Web (web browsing in blogs, social networks, and shopping sites), interactive (chatting applications like Discord and Messenger), and video (watching YouTube and movies). Therefore, the main dataset is composed of several network traffic traces, each belonging to a specific user activity. Then, we filter those pcap files so that there is one source IP and one destination IP for each pcap file. Also, all packets with an empty payload and non-TCP or non-UDP packets should be filtered out.

We use our open-source Montimage security monitoring framework (MMT) [6], in particular "MMT-Probe" module [25] to convert pcap files into csv reports. We can also visualize pcap files with "MMT-Operator" [26]. The .csv files characterize

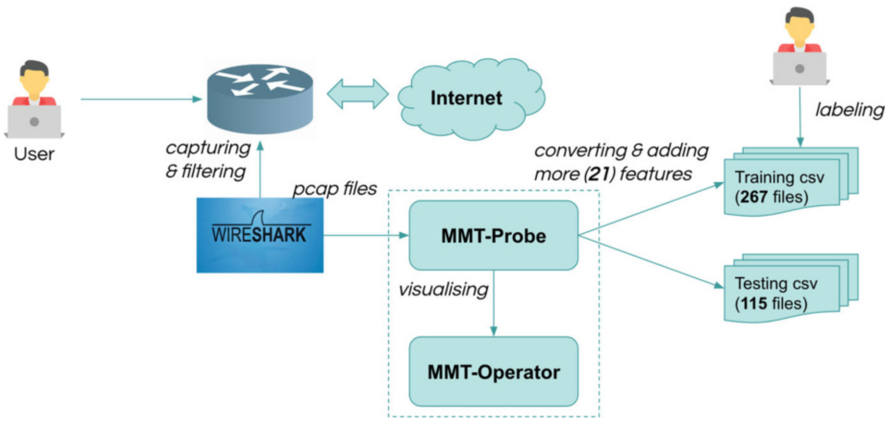


**Fig. 10.6** Traffic dataset generation

the network traffic using the following features: the timestamp, the protocols, the source and destination IP addresses, the payload size in bytes, the number of packets, etc. The full dataset in both pcap and csv formats is being published along with this paper at [23].

#### 10.3.2.4 Dataset Preprocessing

Some activities may produce multiple traces, for instance, MMT-Probe converts a single pcap file capturing a video activity into multiple csv files. Therefore, first of call, we need to merge those csv files belonging to a single activity into a single csv file for further analysis. Next, we only select interesting data concerning the network traffic in the merged csv reports and also compute additional statistics values, like data aggregation. We come up with 21 features that will be described later, as shown in Table 10.7. Finally, the full dataset consists of 382 labeled traces for three traffic classes. The number of traces in Web, interactive, and video activities is 304, 34, and 44, respectively. The final set of processed csv files will be used through the analysis and evaluation of our AI-based classification system.

#### 10.3.2.5 Feature Extraction

Table 10.7 shows 21 features of five main categories. In the group *Duration*, the feature *session_time* shows the total time wherein a user interacts with applications when performing an activity. In the group *Protocol*, there are two features corresponding to the percentage of TCP or UDP traffic. Normally, the two feature values always add up to 100%. In groups *Uplink* and *Downlink*, we use some common features of uplink and downlink communication, such as data

**Table 10.7** Feature selection

| | Category | Feature Id | Feature description |
|---|---|---|---|
| 1 | Duration | *session_time* | Total time wherein a user interacts with apps |
| 2 | Protocol | *%tcp_protocol* | Percentage of TCP traffic |
| 3 | | *%udp_protocol* | Percentage of UDP traffic |
| 4 | Uplink | *ul_data_volume* | Uplink data volume in bytes |
| 5 | | *max_ul_volume* | Maximum of uplink data volume |
| 6 | | *min_ul_volume* | Minimum of uplink data volume |
| 7 | | *avg_ul_volume* | Average of uplink data volume |
| 8 | | *std_ul_volume* | Standard deviation of uplink data volume |
| 9 | | *%ul_volume* | Percentage of uplink data volume |
| 10 | | *nb_uplink_packet* | Number of uplink packets |
| 11 | | *ul_packet* | Percentage of uplink packets |
| 12 | Downlink | *dl_data_volume* | Downlink data volume in bytes |
| 13 | | *max_dl_volume* | Maximum of downlink data volume |
| 14 | | *min_dl_volume* | Maximum of downlink data volume |
| 15 | | *avg_dl_volume* | Average of downlink data volume |
| 16 | | *std_dl_volume* | Standard deviation of downlink data volume |
| 17 | | *%dl_volume* | Percentage of downlink data volume |
| 18 | | *nb_downlink_packet* | Number of downlink packets |
| 19 | | *dl_packet* | Percentage of downlink packets |
| 20 | Speed | *kB/s* | Number of kB per second |
| 21 | | *nb_packet/s* | Number of packets per second |

volume in bytes, number of packets, and percentage of packets to the total packets. In addition, we also compute other useful values, such as the maximum, minimum, mean, and standard deviation. Finally, in the group *Speed*, we add two more features concerning the network speed. This set of features is optimal and suitable to describe network traffic in our scenario.

#### 10.3.2.6 Classification

We implemented the classification application in Python version 3.10 using the open-source popular ML libraries, such as *scikit-learn*,[3] *Keras*,[4] and *TensorFlow 2*.[5] Data preprocessing and postprocessing have been performed using the *numpy* and *pandas* libraries. The graphical plots have been obtained using popular libraries, like *matplotlib* and *seaborn*. For the deep learning model, we used a sequential model as our network consists of a linear stack of layers from the Keras library. Concretely,

[3] https://scikit-learn.org

[4] https://keras.io

[5] https://www.tensorflow.org

a fully connected network structure has three layers: in the first two layers, we used the most widely used activation function, i.e., rectified linear unit (ReLU), while in the output layer we used the Sigmoid activation function. In addition, we used the default models of XGBoost, LightGBM, and random forest for classification.

The full dataset and the code to perform the evaluation of the experiments can be found at [23]. For the evaluation, we randomly split the main dataset, including 382 traces, into the training and testing datasets with a probability of 70% to perform cross validation. Concretely, we used 267 traces to build and train the models and evaluate them against the testing dataset of 115 traces.

## *10.3.3 Evaluation*

### 10.3.3.1 Metrics

We measure the performance of classification models using some popular metrics [27], such as the accuracy, precision, recall (or sensitivity), and F1-score metrics. Those metrics are defined in the following equations, (3), (4), (5), and (6), and are applied to any classification models. More specifically, *TP*, *FP*, and *FN* are the number of true positive instances (correctly classified), the number of false positive instances (incorrectly classified as a class), and the number of false negative instances (incorrectly classified as another class), respectively. The F1-score metric takes both precision and recall into account. All those metrics values are in the range from 0 to 1:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{10.3}$$

$$Precision = \frac{TP}{TP + FP} \tag{10.4}$$

$$Recall = \frac{TP}{TP + FN} \tag{10.5}$$

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall} = \frac{2 \times TP}{2 \times TP + FP + FN} \tag{10.6}$$

### 10.3.3.2 Supervised Classification Models

Table 10.8 contains the metric values of the classification results from the testing dataset for each user activity. Moreover, Fig. 10.7 provides confusion matrices of three classification models to better visualize the relationships between different user activities. While the rows of the confusion matrix illustrate the predicted

**Table 10.8** Experimental results of activity classification for the testing dataset

| | Keras | | | XGBoost | | | LightGBM | | | Random forest classifier | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Activity | Precision | Recall | F1-score | Precision | Recall | F1 | Precision | Recall | F1 | Precision | Recall | F1 |
| Web | 0.98 | 0.99 | 0.98 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 1.00 | 0.99 | 0.99 | 0.99 |
| Interactive | 1.00 | 0.91 | 0.95 | 1.00 | 0.95 | 0.91 | 1.00 | 0.90 | 0.82 | 1.00 | 0.91 | 0.95 |
| Video | 0.78 | 0.70 | 0.74 | 0.88 | 0.78 | 0.70 | 0.91 | 0.95 | 1.00 | 0.91 | 1.00 | 0.95 |
| | Accuracy: 0.95 | | | Accuracy: 0.96 | | | Accuracy: 0.98 | | | Accuracy: 0.98 | | |

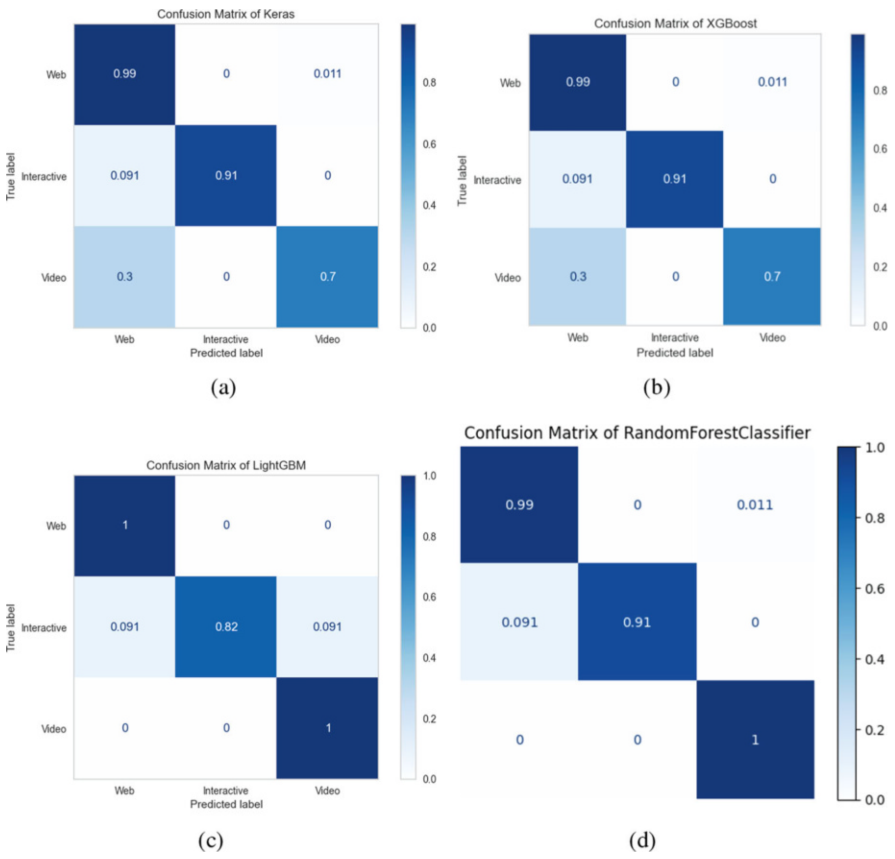


**Fig. 10.7** Random forest classifier model. (**a**) Keras model. (**b**) XGBoost model. (**c**) LightGBM model. (**d**) Random forest classifier model

classification distribution for each user activity, the columns represent the true activity distribution for each predicted class. In addition, the recall for each class is shown in the main diagonal of the confusion matrix.

As shown in Table 10.8 and Fig. 10.7, the accuracy values of four classification models, including the Keras model, the XGBoost model, the LightGBM model, and the random forest classifier model, are outstanding with 95% (5 wrong predictions), 96% (5 wrong predictions), 98% (2 wrong predictions), and 98% (2 wrong predictions), respectively. Other metrics like the precision, recall, and F1-scores are mostly over 95% for all except video activities. Among four classification models, the LightGBM model and the random forest classifier model have the best performance against the testing dataset as they predict correctly all instances of Web and video activities, while the other two models Keras and XGBoost have the same four wrong predictions. Interestingly, for interactive activity classification, the overall best classification model LightGBM performs worst with two wrong predictions, while others have only one. The results suggest that AI classification models are complements to each other and combining those supervised models could give us better results.

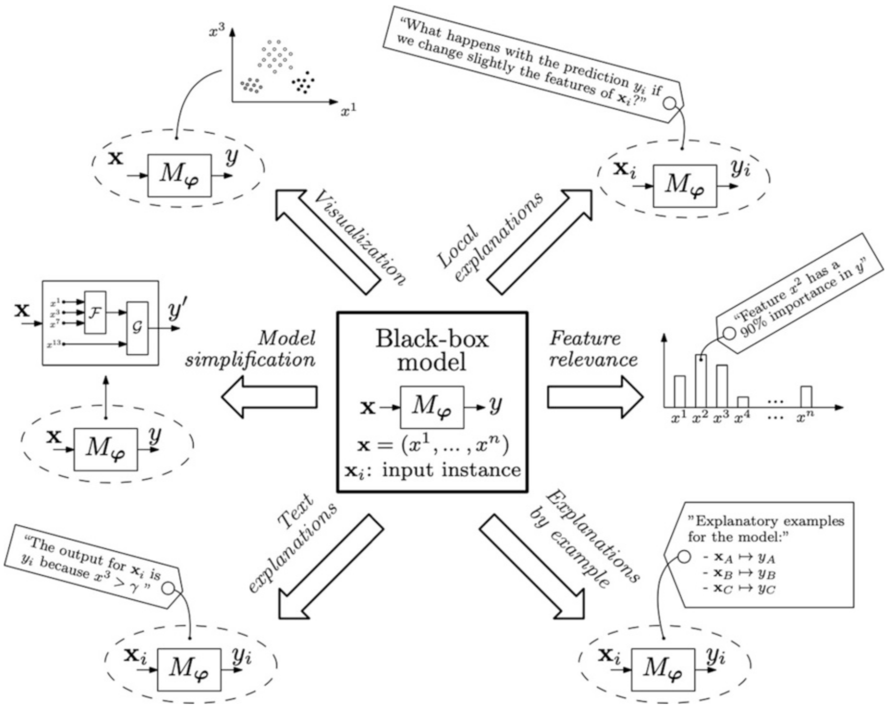
One possible explanation of the worst results of those models in classifying Web and video activities is that users may perform unintentionally those activities at the same time, e.g., user browser web pages that access the content in video activities or some advertising videos pop up on web pages. Furthermore, Web activities have variable behaviors in different forms and share its feature space with other types of user activities.

## *10.3.4 Explainable AI (XAI)*

### **10.3.4.1 State-of-the-Art of XAI Method**

Explainable AI (XAI) [20] is a promising set of technologies that increases the AI black box models' transparency to explain why certain decisions were made. While AI plays a critical role in different domains, XAI is crucial to enhance trust and transparency for people to use future AI-based applications. For instance, in the previous subsections, we employed different AI models for user activity classification and achieve very good results but still incorrectly classify some instances. However, those models are complex with multiple input features and not readily interpretable by design, thus hindering users or even developers to understand and debug them to improve the performance of the AI-based system. Therefore, we need to build an explainability layer on top of the AI models to provide post hoc explainability and enhance their interpretability. As depicted in Fig. 10.8, some popular post hoc explainability methods are *visual explanations*, *local explanations*, *explanations by example*, and *feature relevance explanations*:

- *Local explanations* aim at approximating explanations to less complex solution subspaces for model predictions by only considering a subset of data. *Local Interpretable Model-Agnostic Explanations* (LIME) [22] is a widely popular technique used in interpreting outputs of black box models in several fields and applications.
- *Feature relevance explanations* compute relevance scores of the model features to quantify the contribution or sensitivity of each feature to the model's output. *SHapley Additive exPlanations* (SHAP) [21] is a popular XAI technique that identifies the importance of each feature value in a certain prediction using popular cooperative game theory technique. *Permutation feature importance* is a global XAI method that measures the increase in the prediction error of the model after we permute the feature's tabular values. To assess how important a specific feature is, we compare the initial model with the new model on which the feature's values are randomly shuffled.
- *Explanations by example* consider the extraction of representative data examples that relate to the result generated by a certain model, allowing to get a



Black-box model
$\mathbf{x} \to M_\varphi \to y$
$\mathbf{x} = (x^1, \dots, x^n)$
$\mathbf{x}_i$: input instance
Visualization
Local explanations
"What happens with the prediction $y_i$ if we change slightly the features of $\mathbf{x}_i$?"
Model simplification
Feature relevance
"Feature $x^2$ has a 90% importance in $y$"
Text explanations
"The output for $\mathbf{x}_i$ is $y_i$ because $x^3 > \gamma$"
Explanations by example
"Explanatory examples for the model:"
- $\mathbf{x}_A \mapsto y_A$
- $\mathbf{x}_B \mapsto y_B$
- $\mathbf{x}_C \mapsto y_C$

**Fig. 10.8** Conceptual diagram showing different existing post hoc explainability methods for ML models [20]

better understanding of the model. Some XAI methods of this category are *counterfactual explanations* [28] and *adversarial examples*.

Each of those techniques covers a way in which humans explain an object and the combination of all methods provides us the whole explanations about the AI models. However, as many methods may be suitable for different types of the AI models or datasets, we need to consider the best appropriate methods for the concrete problem being solved. Next, we apply some popular XAI methods, such as SHAP and LIME, to provide both global and local explanations of the AI models that we used previously for user network activity classification. Since both SHAP and LIME are model-agnostic XAI methods, which imply that they can be applied to any ML models, we will discuss in details the explanations for the Keras model. The full results can be found at [23].

#### 10.3.4.2 SHAP

Lundberg et al. proposed the SHapley Additive exPlanations (SHAP) method which offers a high level of interpretability for a model [21]. The SHAP values, which are based on the concepts of game theory, provide both *global* and *local* explainability of any ML models. For global explainability, the SHAP values show how much each input feature contributes, either positively or negatively, to the model's global output. For local explainability, as each prediction has its own set of SHAP values, we can explain why the model makes a specific prediction and input feature importance. Our implementation uses the *KernelExplainer* method of the SHAP library [35] to calculate SHAP values and build summary and dependence plots. Specifically, the *KernelExplainer* builds a weighted linear regression to compute the variable importance values using the dataset, the labeled outputs, and the model predictions. In addition, we apply also the dedicated method *DeepExplainer* that performs calculations of the SHAP values for the Keras model faster than the previous one *KernelExplainer*.

*SHAP Summary Plots* They show the positive and negative relationships of the AI models with its outcome. Figure 10.9 shows SHAP summary plots for Web, interactive, video, and all activity classification using the Keras model. The summary plot consists of many dots representing instances of the dataset. Vertical location shows the input features that are ranked in descending order in terms of feature importance. The horizontal location shows whether the effect of a single feature is associated with a higher or lower model prediction. Color illustrates whether that feature has a high (in red) or low (in blue) impact on that prediction. As depicted in Fig. 10.9a, the feature $\%tcp\_protocol$ has a positive and high impact on predicting an instance as a Web activity because of a large number of red dots on the x-axis. Similarly, we can say the feature $\%udp\_protocol$ is negatively correlated with Web activities but highly contributed to predict interactive and video activities, as shown in Fig. 10.9b, c. Furthermore, from Fig. 10.9d, we observe that the most five important features contributing globally to our AI-based classification are $\%tcp\_protocol$, $\%udp\_protocol$, $nb\_downlink\_packet$, $dl\_packet$, and $ul\_data\_volume$.

*SHAP Dependence Plots* They show the effect of a single input feature across the whole dataset. Figure 10.10 shows some interesting SHAP dependence plots for Web activity classification using the Keras model. Each dot in the plot represents a single prediction from the dataset. The x-axis and y-axis show the values of an input feature from the dataset and its SHAP values underlying how much this feature has contributed to the prediction, respectively. Similar to SHAP summary plots, the color of SHAP dependence plots corresponds to an interaction effect, like high in red and low in blue, between the input feature we are plotting and the second feature. For instance, the SHAP dependence plot of the feature $session\_time$ shows that this feature interacts mostly with the feature $nb\_downlink\_packet$, as depicted in Fig. 10.10a. In addition, Fig. 10.10b shows that there is an approximately linear and negative trend between the feature $nb\_uplink\_packet$ and the second feature $\%udp\_protocol$ that interacts with $nb\_uplink\_packet$ frequently. Interestingly, from Fig. 10.10c, d, the most two important features $\%tcp\_protocol$ and $\%udp\_protocol$ interact most with other features of the *Downlink* category, such as $\%dl\_volume$ and $dl\_data\_volume$. This also explains why the *Downlink*

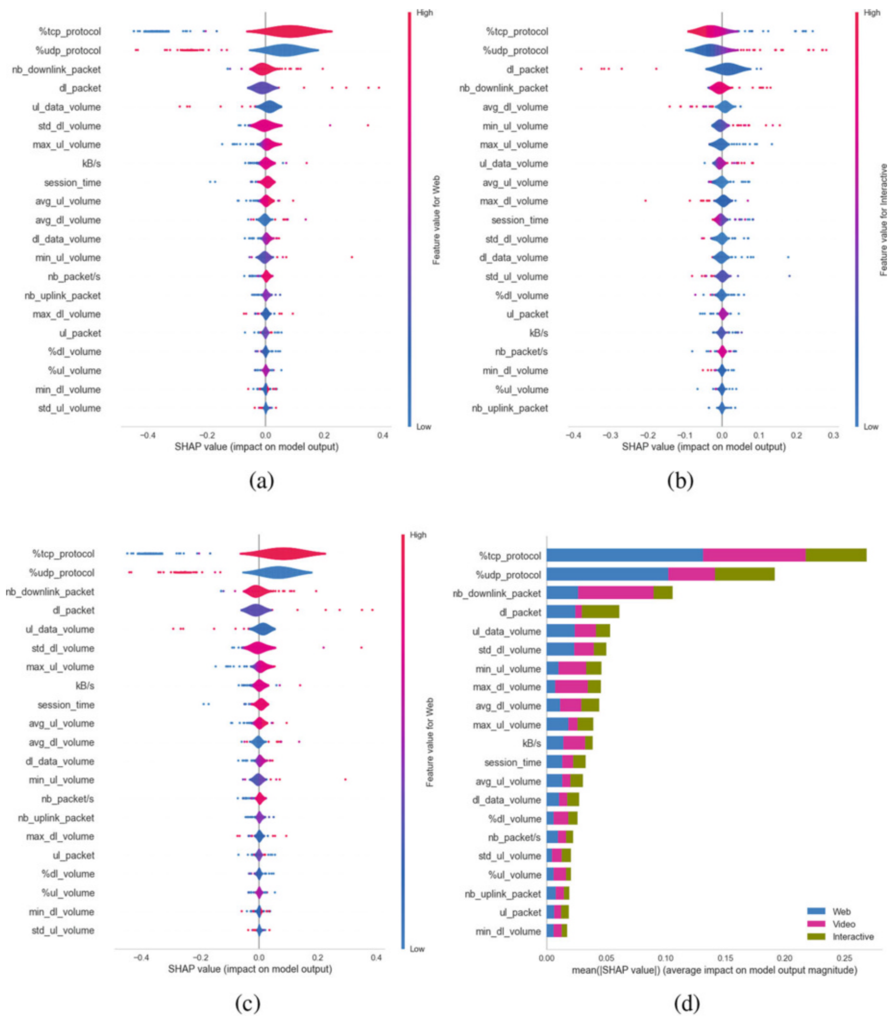


**Fig. 10.9** SHAP summary plots for the feature importance of the Keras model. (**a**) Web activity. (**b**) Interactive activity. (**c**) Video activity. (**d**) All three activities

features have great impact on the final prediction of the Keras model for user activity classification, as discussed above.

### 10.3.4.3 LIME

Ribeiro et al. proposed Local Interpretable Model-Agnostic Explanations (LIME) method that aims to explain individual predictions of black box AI models. While

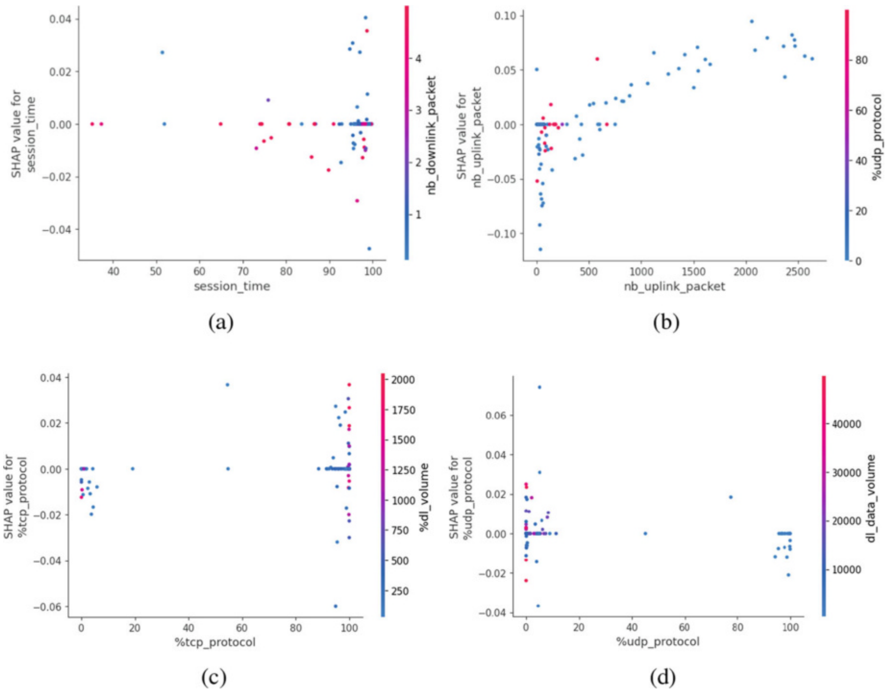


**Fig. 10.10** SHAP dependence plots for Web activity classification using the Keras model

the SHAP values of a feature represent their contribution to one or several sets of features, LIME aims to provide *local* explainability that are locally faithful within the surroundings or vicinity of the sample data being explained. The LIME method is compatible with many different classifiers and can be used with image, tabular, and text data. Similar to the SHAP method, LIME does not take the model into account and thus can be applied to any models. Our implementation uses the *LimeTabularExplainer* method of the LIME library [34] to calculate values and build plots.

Figure 10.11 shows the explanation for a single instance from the testing dataset. The leftmost values are the prediction probabilities of our classifier, that in this case is the Keras model. Concretely, the Keras model predicts correctly this particular instance as Web activity with 100% of confidence. The numbers on the right reflect the average influence of that particular feature value in the final prediction, for example, as a Web/interactive/video activity or not. This set of values encapsulates the behavior of the LIME's linear model in the neighborhood of the sample data that we try to explain.

Given the training dataset, having $ul\_packet > 0.84$, $min\_dl\_volume <= 0.12$, and $\%tcp\_protocol > 0.47$ would increase on average the prediction probability of that instance being a Web activity by 0.13, 0.13, and 0.10, respectively. Note that the concrete features' values are in the table in the bottom left. Moreover,

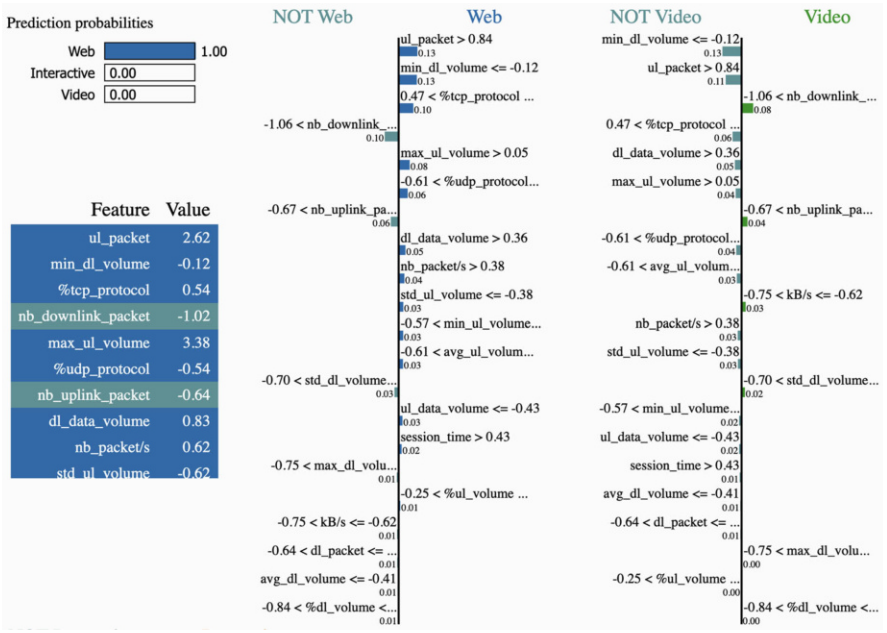


**Fig. 10.11** Illustration of the LIME method results for the prediction using the Keras model (details of *not interactive* and *interactive* are omitted)

having $nb\_downlink\_packet > -1.06$ would decrease on average the prediction probability by 0.10. We have the similar observation by looking at the positive impact of the input feature $nb\_downlink\_packet$ with the value −1.02 on the column *Video*. This is because $nb\_downlink\_packet$ is the second most important feature in classifying an instance as a video activity, as discussed earlier in Fig. 10.9c showing the SHAP summary plot for video activities. Overall, by only looking at the two columns *NOT Web* and *Web*, we observe that there are more input features with a bigger positive contribution to the prediction probability of this instance being a Web activity. This is why the Keras model gives us a prediction of being a Web activity for this particular instance. We can also conclude that two XAI methods SHAP and LIME are complement to each other and provide us the similar explanations of our AI-based application for user activity classification.

#### 10.3.4.4 Shapash

Shapash [29] is an open-source Python library to visualize AI models to make them reliable, transparent, and understandable for everyone. It is compatible with many models, including scikit-learn, XGBoost, and LightGBM models for both classification and regression tasks. Moreover, Shapash allows users to easily understand their AI models via a nice and user-friendly Web dashboard to navigate between the feature importance and global and local explainability with popular XAI methods like SHAP, Active Coalition of Variables[6] (ACV), and LIME as backend. As depicted in Fig. 10.12, on the Shapash's dashboard of the random forest classifier model, we can easily visualize and interact with the dataset and observe how each feature contributes to model predictions and local explanation of an individual instance.

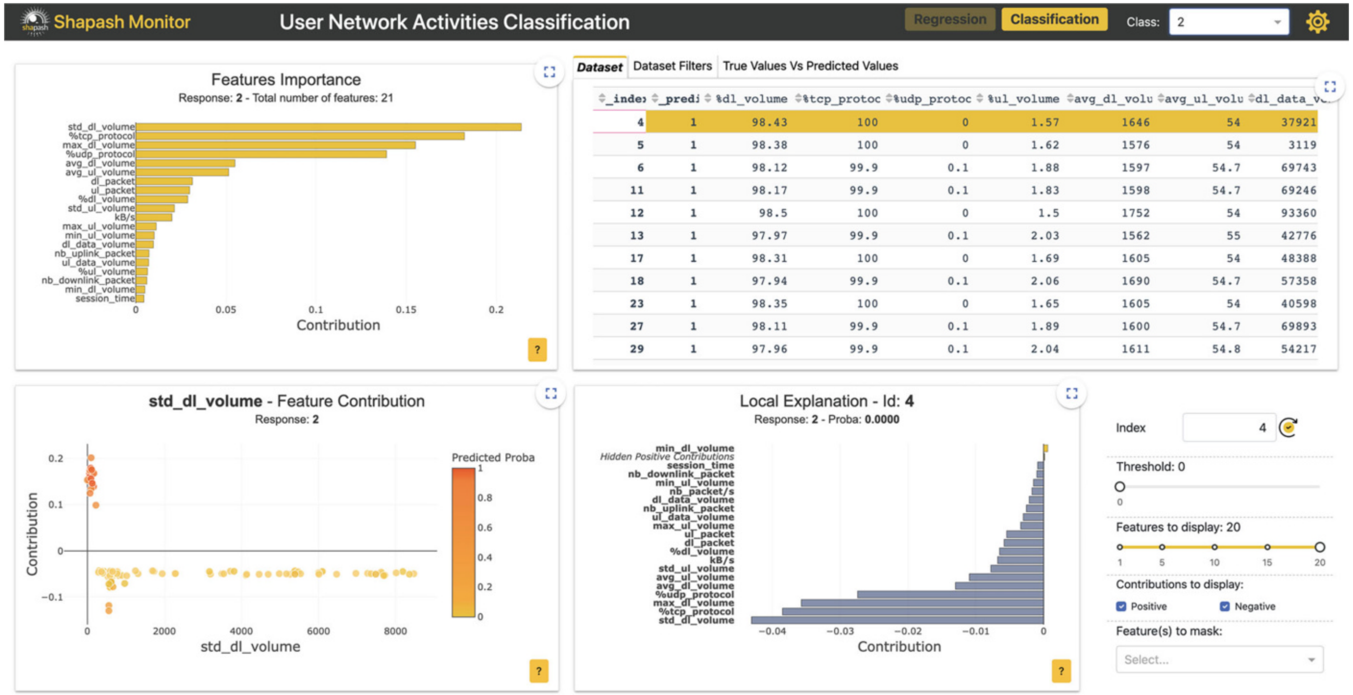
Different XAI methods may give us different results or different explanations. To increase the degree of confidence of applying different XAI methods, we need to define some metrics to assess the quality of their explanations. Shapash can measure some interesting metrics [30] to assess the degree of confidence on different XAI methods as follows:

- The *consistency metric* compares different XAI methods and evaluates them to see how close the explanations are to each other, for example, by calculating an average distance between the explainability methods. If different XAI methods lead to similar results, this would mean a higher degree of confidence can be placed in using them. If not, we would need to carefully interpret the explanations of each method to identify which one is the best.
- The *stability metric* evaluates the similarity between different instances under two criteria: those instances must be close in the feature space and have similar outputs. If instances are similar, we would expect the respective model output for these instances to be similar as well. Therefore, this metric allows for building trust in a specific explanation.
- The *compacity metric* seeks to reduce complexity and overexplaining by measuring the explainability of a decision in relation to only the most important features. For each instance, after identifying feature importance using XAI methods, we select a subset of features with the highest contributions and observe how well they approximate the model.

However, Shapash has still some limitations: First, users may need to develop new or less popular XAI methods that have not been supported by Shapash. Second, multi-class classification, like our user activity classification problem, is not supported yet to compute some metrics discussed above. Therefore, we *simplify* our problem for *classifying an instance as a web activity or not*. Herein, we can employ the Shapash library to calculate some confidence metrics of the random forest classifier model.

The consistency plots in Fig. 10.13 show the average distances between different XAI methods and different types of the SHAP method. Clearly, SHAP and LIME are more similar than ACV as the average distance between SHAP and LIME is smallest. In addition, Kernel SHAP and sampling SHAP produce more similar explanations across all the features. Also, this metric extracts five real comparisons from the dataset with distances similar to those in the average distance plot.

[6] https://github.com/salimamoukou/acv00



| _inde | _predi | %dl_volume | %tcp_protoc | %udp_protoc | %ul_volume | avg_dl_volu | avg_ul_volu | dl_data_v |
|---|---|---|---|---|---|---|---|---|
| 4 | 1 | 98.43 | 100 | 0 | 1.57 | 1646 | 54 | 37921 |
| 5 | 1 | 98.38 | 100 | 0 | 1.62 | 1576 | 54 | 3119 |
| 6 | 1 | 98.12 | 99.9 | 0.1 | 1.88 | 1597 | 54.7 | 69743 |
| 11 | 1 | 98.17 | 99.9 | 0.1 | 1.83 | 1598 | 54.7 | 69246 |
| 12 | 1 | 98.5 | 100 | 0 | 1.5 | 1752 | 54 | 93360 |
| 13 | 1 | 97.97 | 99.9 | 0.1 | 2.03 | 1562 | 55 | 42776 |
| 17 | 1 | 98.31 | 100 | 0 | 1.69 | 1605 | 54 | 48388 |
| 18 | 1 | 97.94 | 99.9 | 0.1 | 2.06 | 1690 | 54.7 | 57358 |
| 23 | 1 | 98.35 | 100 | 0 | 1.65 | 1605 | 54 | 40598 |
| 27 | 1 | 98.11 | 99.9 | 0.1 | 1.89 | 1600 | 54.7 | 69893 |
| 29 | 1 | 97.96 | 99.9 | 0.1 | 2.04 | 1611 | 54.8 | 54217 |

**Fig. 10.12** Shapash's dashboard for visualizing the random forest classifier model

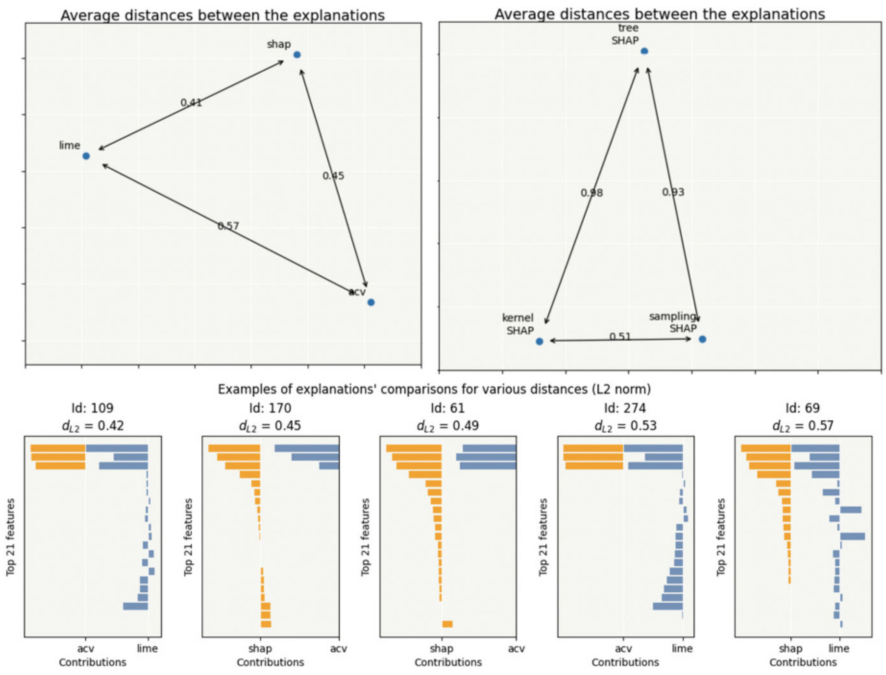


**Fig. 10.13** Consistency of explanations provided by different XAI methods

The compacity plots in Fig. 10.14 show the link between the level of approximation, the number of required features to reach it, and the proportion of the dataset on which it works. In the left graph, top *nine* most important features explain at least 90% of the model for 100% of the instances. In this case, considering a small subset of features could provide a reliable explanation for almost all instances. In the right graph, top five features reach at least 80% of the reference model for 100% of the instances. Therefore, if we want more precise explanations, we would need to consider more than top five features in the explanations.

Figure 10.15 is the stability plot showing the neighborhood in terms of features and model's output around each particular instance. The x-axis and y-axis show the average variability of the feature across the instances' neighborhood and the average importance of the feature across the dataset, respectively. Consequently, left features are much more stable in the neighborhood than right ones, and top features are more important than bottom ones. As shown in Fig. 10.15, $\%tcp\_protocol$, $std\_dl\_volume$, $max\_dl\_volume$, and $\%udp\_protocol$ are important features and have strong and relatively stable contributions to the model's output. On the other hand, some features that belong to the *Uplink* category, such as $\%ul\_volume$, $nb\_uplink\_packet$, and $ul\_data\_volume$, are unstable; thus, we should be careful to interpret explanations around these features.

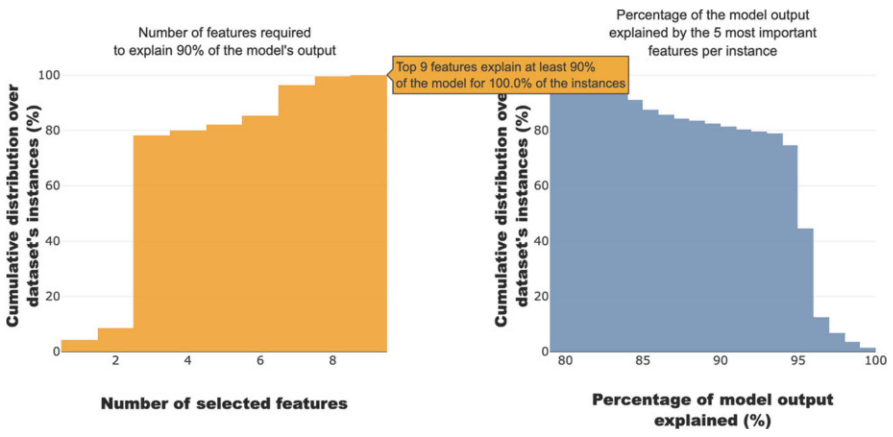


**Fig. 10.14** Compacity of explanations

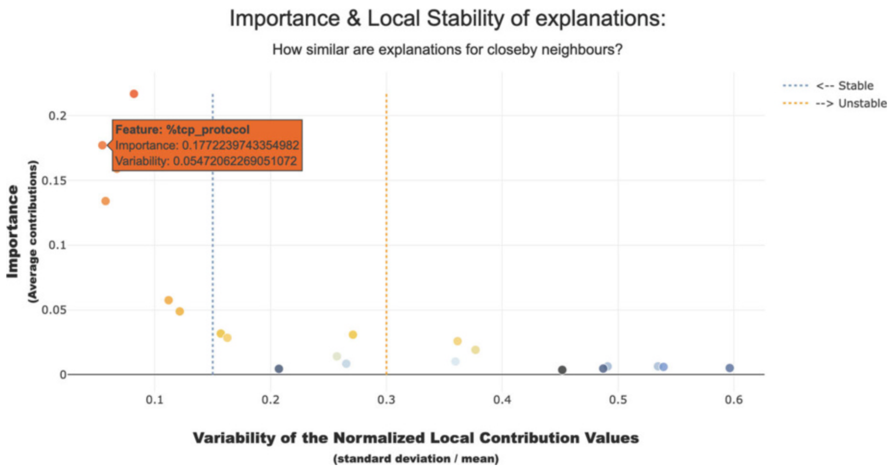


**Fig. 10.15** Importance and local stability of explanations

## 10.4 Discussion

XAI (explainable artificial intelligence) has recently gained a lot of attention as it provides insights into the black box nature of many machine learning models. In the context of network classification, XAI can be used to provide explanations for the predictions made by the network. This can help users understand why a particular prediction was made, which can be useful for debugging the network or improving its accuracy.

XAI-based anomaly detection helps address concerns around false positives and false negatives, since the algorithm can provide insights into why certain activity is classified as anomalous, and can be refined over time to improve its accuracy.

However, XAI-based anomaly detection also requires significant expertise and resources to develop and maintain. It involves working with large datasets and complex statistical models and requires a deep understanding of both network security and machine learning. Therefore, it is important to carefully evaluate the costs and benefits of XAI-based anomaly detection before implementing it in a network environment. More work on this cost assessment is planned by the authors of this chapter.

### *10.4.1 Conclusion and Future Work*

In this chapter, we present our AI-based application for anomaly detection and activity classification based on network traffic. We employ and evaluate different supervised learning classification models, such as random forest, Keras, XGBoost, and LightGBM against our full dataset. The best model is LightGBM with up to 98% global accuracy. Furthermore, we provide both global and local explanations of our evaluated models using popular XAI methods, like SHAP and LIME, to have deeper insights into the dataset and the models' predictions in our scenario.

As an extension of this work, we will try to improve our classification system by considering more complex (hybrid) ML models, adding more input features and taking advantage of the complementarity of different XAI methods to extend the existing interpretability analysis. The extra explainability layer could be useful for different AI-based applications, such as root cause analysis [31] or our advanced encrypted traffic analysis [8]. We also aim to produce larger datasets with more types of activities, such as data transfer, idle behavior, or simultaneous activities. Some other future work can approach different security scenarios in which we may need to identify specific security applications rather than general network traffic, for example, intrusion detection, malware detection, and different types of malware classification. In addition, we could use our 4G/5G testbeds [32] to generate real datasets for mobile user activity classification similar to [33].

**Acknowledgments** We thank Viet Pham and Hadil Saadaoui for help with experimental setup. This work has received funding from the European Union's H2020 Programme under Grant Agreement No. 957212 for the VeriDevOps project as well as Grant Agreement No. 101021808 for the SPATIAL project.

## References

1. S. Eltanbouly, M. Bashendy, N. AlNaimi, Z. Chkirbene, A. Erbad, Machine learning techniques for network anomaly detection: a survey, in *The IEEE International Conference on Informatics, IoT, and Enabling Technologies, ICIoT 2020*, Doha, Qatar, 2–5 Feb 2020 (IEEE, 2020), pp. 156–162. https://doi.org/10.1109/ICIoT48696.2020.9089465

2. S. Wang, J.F. Balarezo, S. Kandeepan, A. Al-Hourani, K.G. Chavez, B. Rubin-stein, Machine learning in network anomaly detection: a survey. IEEE Access **9**, 152379–152396 (2021). https://doi.org/10.1109/ACCESS.2021.3126834
3. G. Apruzzese, P. Laskov, E.M. de Oca, W. Mallouli, L.B. Rapa, A.V. Grammatopoulos, F.D. Franco, *The Role of Machine Learning in Cybersecurity*. CoRR abs/2206.09707 (2022). https://doi.org/10.48550/arXiv.2206.09707
4. M. Bahri, F. Salutari, A. Putina, M. Sozio, AutoML: state of the art with a focus on anomaly detection, challenges, and research directions. Int. J. Data Sci. Anal. **14**(2), 113–126 (2022). https://doi.org/10.1007/s41060-022-00309-0
5. R. Dwivedi, D. Dave, H. Naik, S. Singhal, O.F. Rana, P. Patel, B. Qian, Z. Wen, T. Shah, G. Morgan, R. Ranjan, Explainable AI (XAI): core ideas, techniques, and solutions. ACM Comput. Surv. **55**(9), 194:1–194:33 (2023). https://doi.org/10.1145/3561048
6. Montimage tools. https://github.com/Montimage
7. I. Ghafir, V. Prenosil, J. Svoboda, M. Hammoudeh, A survey on network security monitoring systems, in *4th IEEE International Conference on Future Internet of Things and Cloud Workshops, FiCloud Workshops 2016*, Vienna, Austria, 22–24 Aug 2016, ed. by M. Younas, I. Awan, J.E. Haddad (IEEE Computer Society, 2016), pp. 77–82. https://doi.org/10.1109/W-FiCloud.2016.30
8. Montimage advanced encrypted traffic analysis tool. https://github.com/Montimage/acas
9. Z. Salazar, A.R. Cavalli, W. Mallouli, F. Sebek, F. Zaïdi, M.E. Rakoczy, Monitoring approaches for security and safety analysis: application to a load position system, in *The 15th IEEE International Conference on Software Testing, Verification and Validation Workshops ICST Workshops 2022*, Valencia, Spain, 4–13 Apr 2022 (IEEE, 2022), pp. 40–48. https://doi.org/10.1109/ICSTW55395.2022.00021
10. CSE-CIC-IDS2018 dataset on AWS. A collaborative project between the Communications Security Establishment (CSE) & the Canadian Institute for Cybersecurity (CIC). https://www.unb.ca/cic/datasets/ids-2018.html
11. T. Silhan, S. Oehmcke, O. Kramer, Evolution of stacked autoencoders, in *The IEEE Congress on Evolutionary Computation, CEC 2019*, Wellington, New Zealand, 10–13 June 2019 (IEEE, 2019), pp. 823–830. https://doi.org/10.1109/CEC.2019.8790182
12. K. O'Shea, R. Nash, *An Introduction to Convolutional Neural Networks*. CoRR abs/1511.08458 (2015)
13. Simple and efficient tools for predictive data analysis. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html
14. L. Ding et al., A classification algorithm for network traffic based on improved support vector machine. J. Comput. **8**(4), 1090–1096 (2013)
15. M. Lotfollahi et al., Deep packet: a novel approach for encrypted traffic classification using deep learning. Soft Comput. **24**(3), 1999–2012 (2020)
16. Y. Liu, W. Li, Y. Li, Network traffic classification using K-means clustering, in *Second International Multi-symposiums on Computer and Computational Sciences (IMSCCS 2007)* (IEEE, 2007)
17. R. Bar-Yanai et al., Realtime classification for encrypted traffic, in *International Symposium on Experimental Algorithms* (Springer, Berlin/Heidelberg, 2010)
18. B. Saltaformaggio et al., Eavesdropping on fine-grained user activities within smartphone apps over encrypted network traffic, in *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, 2016
19. V. Labayen et al., Online classification of user activities using machine learning on network traffic. Comput. Netw. **181**, 107557 (2020)
20. A.B. Arrieta, N. Díaz-Rodríguez, J. Del Ser, A. Bennetot, S. Tabik, A. Barbado, S. García et al., Explainable Artificial Intelligence (XAI): concepts, taxonomies, opportunities and challenges toward responsible AI. Inf. Fusion **58**, 82–115 (2020)
21. S.M. Lundberg, S.-I. Lee, A unified approach to interpreting model predictions. Adv. Neural Inf. Proces. Syst. **30**, 4768–4777 (2017)

22. M.T. Ribeiro, S. Singh, C. Guestrin, "Why should i trust you?" Explaining the predictions of any classifier, in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016
23. Montimage, Full dataset and code of explaining ML models for user network activities classification. https://github.com/Montimage/activity-classification
24. Wireshark. https://www.wireshark.org/
25. MMT-Probe: a high-performance network monitoring tool. https://github.com/Montimage/mmt-probe
26. MMT-Operator. https://github.com/Montimage/mmt-operator
27. Scikit-learn metrics and scoring: quantifying the quality of predictions. https://scikit-learn.org/stable/modules/model_evaluation.html
28. S. Wachter, B. Mittelstadt, C. Russell, Counterfactual explanations without opening the black box: automated decisions and the GDPR. Harv. JL Tech. **31**, 841 (2017)
29. Shapash makes Machine Learning models transparent and understandable by everyone. https://github.com/MAIF/shapash.
30. Shapash tutorial: Building confidence on explainability methods. https://shapash.readthedocs.io/en/latest/tutorials/explainability_quality/tuto-quality01-Builing-confidence-explainability.html
31. L. Nguyen, V.H. La, W. Mallouli, E. Montes de Oca, Validation, verification and root-cause analysis, in *Devops for Trustworthy Smart IOT Systems*, vol. 173, 2021
32. M.-D. Nguyen, V.H. La, R. Cavalli, E.M. de Oca, Towards improving explainability, resilience and performance of cybersecurity analysis of 5G/IoT networks (work-in-progress paper), in *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2022, pp. 7–10. https://doi.org/10.1109/ICSTW55395.2022.00016
33. A. Nascita et al., XAI meets mobile traffic classification: understanding and improving multimodal deep learning architectures. IEEE Trans. Netw. Serv. Manag. **18**(4), 4225–4246 (2021)
34. Lime: Explaining the predictions of any machine learning classifier. https://github.com/marcotcr/lime
35. SHAP: A game theoretic approach to explain the output of any machine learning model. https://github.com/slundberg/shap