UNIVERSITÉ DU LUXEMBOURG

PhD-FSTM-2020-44
The Faculty of Sciences, Technology and Medicine

DISSERTATION

Defence held on 08/09/2020 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Xuân Phú MAI
Born on 15th October 1985 in Binh Dinh (Vietnam)

AUTOMATED, REQUIREMENTS-BASED SECURITY TESTING OF WEB-ORIENTED SOFTWARE SYTEMS

Dissertation defence committee

Dr. Ing. Lionel C. Briand, Dissertation Supervisor
Professor, University of Luxembourg, Luxembourg

Dr. Fabrizio Pastore, Chairman
Associate Professor, University of Luxembourg, Luxembourg

Dr. Seung Yeob Shin, Vice Chairman
Research associate, University of Luxembourg, Luxembourg

Dr. Mariano Ceccato,
Assistant Professor, University of Verona, Italy

Dr. Sergio Segura Rueda,
Associate Professor, University of Seville, Spain

Abstract

Motivation and Context. Modern Internet-based services (e.g., home-banking, personal-training, healthcare) are delivered through Web-oriented software systems which run on multiple and different devices including computers, mobile devices, wearable devices, and smart TVs. They manage and exchange users' personal data such as credit reports, locations, and health status. Therefore, the security of the system and its data are of crucial importance.

Unfortunately, from security requirements elicitation to security testing, there are a number of challenges to be addressed to ensure the security of Web-oriented software systems. First, existing practices for capturing security requirements do not rely on templates that ensure the specification of requirements in a precise, structured, and unambiguous manner. Second, security testing is usually performed either manually or is only partially automated. Most of existing security testing automation approaches focus only on specific vulnerabilities (e.g., buffer overflow, code injection). In addition, they suffer from the oracle problem, i.e., they cannot determine that the software does not meet its security requirements, except when it leads to denial of service or crashes. For this reason, security test automation is usually partial and only addresses the generation of inputs and not the verification of outputs.

Though, in principle, solutions for the automated verification of functional requirements might be adopted to automatically verify security requirements, a number of concerns remain to be addressed. First, there is a lack of studies that demonstrate their applicability, in the context of security testing. Second, the oracle problem remains an open problem in many aspects of software testing research, not only security testing. In the context of functional testing, metamorphic testing has shown to be a viable solution to address the oracle problem; however, it has never been studied in the context of security testing.

Contributions. In this dissertation, we propose a set of approaches to address the above-mentioned challenges. (1) To model security requirements in a structured and analyzable manner, we propose a use case modeling approach that relies on a restricted natural language and a template already validated in the context of functional testing. It introduces the concepts of security use case specifications (i.e., what the system is supposed to do) and misuse case specifications (i.e., malicious user behaviours that the system is supposed to prevent). Moreover, we propose a template for capturing guidelines for the mitigation of security threats. (2) To verify that systems meet their security requirements, we propose an approach to automatically generate security test cases from misuse use case specifications. More precisely, we propose a natural language programming solution that automatically generates executable security test cases and test inputs from misuse case specifications in natural language. (3) To address the oracle problem, we propose a metamorphic testing solution for Web-oriented software systems. The solution relies on a predefined set of metamorphic relations that capture (a) how an attacker likely alters a valid input to exploit a vulnerable system and (b) how the output of the system should change as a result of the attack if the system meet its security requirements. Our solution relies on Web-crawlers to automatically identify the valid inputs to be used for testing. (4) We identify a set of testability guidelines to facilitate the adoption of the proposed approaches in software projects. The identified guidelines indicate (a) which types of vulnerabilities can be addressed through the solutions proposed in this dissertation and (b) which design solutions should be integrated into the system to enable effective test automation.

Acknowledgements

I would like to state my sincere gratitude to my supervisor, Professor Lionel Briand, for his invaluable support, comments and counsel throughout the whole PhD project. This work would not have gone far without his guidance, enthusiasm, and encouragement.

I would like to express my infinite gratitude to my co-supervisor, Associate Professor Fabrizio Pastore, for his guidance, inspiration, corrections and patience, which leads to the success of my work. I have learned a lot from him both professionally and personally.

I would also like to thank Arda Göknil for his care, encouragement, and advice. Moreover, I appreciate that Lwin Khin Shar instructed me during the first year of my PhD.

I am grateful to the members of my defence committee: Seung Yeob Shin, Mariano Ceccato, and Sergio Segura Rueda for their attendance, insightful suggestions and remarks at my defence.

I am so thankful to all my colleagues in the Software Verification and Validation Lab for creating an inspiring and welcoming working environment.

Last but not least I would like to thank my family for always supporting me to achieve and pursue all my goals and dreams. I am completely grateful with Kieu Oanh, my life companion, and Kieu Thu, my sweet daughter, who have been by my side every day of this journey.

Luxembourg National
Research Fund

erc

Supported by the Fonds National de la Recherche, Luxembourg (INTER/AAL/15/11213850), and the European Research Council (ERC/694277).

Contents

List of Figures

List of Tables

List of Acronyms

API Application Programming Interface.

CAPEC Common Attack Pattern Enumeration and Classification.
CNP CogComp NLP pipeline.
CVE Common Vulnerabilities and Exposures.
CWE Common Weakness Enumeration.

MCP Misuse Case Programming.
MR Metamorphic Relation.
MST Metamorphic Security Testing.
MT Metamorphic Testing.

NL Natural Language.
NLP Natural Language Processing.
NLRP Natural Language Programming.

OWASP The Open Web Application Security Project®.

RMCM Restricted Misuse Case Modeling.
RMCM-V Restricted Misuse Case Modeling - Verifier.
RQ Research Question.
RUCM Restricted Use Case Modeling.

SMRL Security Metamorphic Relation Language.
SQLI SQL Injection.
SRL Semantic Role Labeling.

TF Testability Feature.

UML Unified Modeling Language.

XSS Cross Site Scripting.

Chapter 1

Introduction

1.1 Context

Modern Internet-based services are delivered through *web-oriented software systems* [Gall, 2008] that include multiple and different devices, e.g., computers, mobile devices, wearable devices, smart TVs. Typical services of this type are healthcare [Sunwave, 2020], home-banking [USC Credit Union, 2017], personal-training [FitBit, 2017], music-streaming [Spotify, 2017], and food-delivery [DeliveryHero, 2017]. The common characteristic of these systems is that they manage and exchange users' personal data such as locations, credit reports, and health status. Therefore, the security of the system and its data are of crucial importance.

Multiple case study systems that are representative of modern Internet-based services are considered in this dissertation. One case study is the EDLAH2 [EDLAH2, 2017a] system, which is the system developed in the context of a European project [EDLAH2, 2017b]. This project aims to rely on gamification principles to engage elder people in improving the quality of their life, by providing them with daily objectives such as walking, communicating with other people, or playing games. EDLAH2 is based on a platform including multiple devices (e.g., computers, tablets, bracelets) and multiple services (e.g., collecting daily activities, playing games, communication). Other systems are Jenkins [Eclipse Foundation, 2020], which is a system used to deliver ICT services, and Joomla! [Joo, 2020], a content management system used as backbone for popular websites [Nintendo, 2020, Data2.eu, 2020].

This dissertation focuses on the definition of methods to ensure that *web-oriented software systems* meet their security requirements. To achieve this objective, it is necessary, first, to define strategies for eliciting security requirements in a structured and analyzable form to support communication among stakeholders and ease the security verification process, while accounting for the specificities of Web-oriented software systems. Second, it is necessary to identify methods that verify that all the security requirements are actually addressed in the implemented system. This may be achieved through test automation since it is more scalable than other verification approaches (e.g., based on formal verification). Third, to lower testing costs, it is necessary to identify means to automatically generate test inputs and test oracles [Barr et al., 2015, Pezze and Zhang, 2014].

To specify the security requirements of Web-oriented software systems, the characteristics of the developed services and the device types on which the services will be deployed need to be specified. Some approaches have been proposed to elicit and analyze security requirements in terms of use cases such as abuse cases [McDermott and Fox, 1999, McDermott, 2001], security use cases [Firesmith, 2003], and misuse cases [Opdahl and Sindre, 2009, Rostad, 2006, Sindre and Opdahl, 2005, Sindre and Opdahl, 2001]. However, the applicability of these approaches in the security context is limited; indeed, there is a lack of methods for the definition of threat scenarios and mitigation schemes for these threats. In this dissertation, we propose, apply, and assess a use case modeling approach to model the security requirements of Web-oriented software systems.

After eliciting software security requirements in natural language in terms of positive requirements (i.e., what the system is supposed to do to ensure security) and negative requirements (i.e., undesirable behavior undermining security), it is necessary to validate whether positive requirements are properly addressed by the system to prevent the scenarios captured by negative requirements. In this dissertation, we address the problem of automatically generating executable security test cases from security requirements in natural language to verify if Web-oriented software systems comply with them. Since existing approaches for the generation of test cases from natural language (NL) requirements verify only positive requirements [Carvalho et al., 2014, de Figueiredo et al., 2006, Wang et al., 2015a, Wang et al., 2015b, Kaplan et al., 2008], we focus on the problem of generating test cases from negative requirements. To this end, we propose, apply and assess a natural language programming [Manning et al., 2014, Le et al., 2013, Guzzoni et al., 2007, Landhausser et al., 2017, Thummalapenta et al., 2012] approach that automatically generates security test cases from misuse case specifications. A *misuse case specification* is a description, in natural language, of the actions performed by a malicious user to trigger a behaviour that is not legal, based on the security requirements of the system. A misuse case specification thus captures an *attack* against the system. The specific fault causing the software not to comply with its security requirements is called a *vulnerability*. Vulnerabilities might share common characteristics, i.e., types of vulnerabilities. In this dissertation, in accordance with common practice [CWE, 2020s], we use the term *weakness* to indicate a vulnerability type.

Approaches for the generation of test cases are beneficial when they (1) enable the selection of the test inputs to be used for testing, (2) lead to the generation of executable test cases that can automatically exercise the software under test, (3) are capable of automatically deriving test oracles (i.e, means to determine if the software outputs match the requirements, based on the provided inputs). Unfortunately, although test input generation and test cases executions are addressed by many state-of-the-art work, the automated generation of test oracles is an open research problem in many contexts [Barr et al., 2015]. In many systems, a test oracle may not exist or may not be easy to specify, even manually, because of the many inputs to be tested. For instance, the security test case for the bypass authorization vulnerability should check, for every user role in the system, whether it is possible to access resources dedicated to another user role [Meucci and Muller, 2014]. To alleviate the oracle problem, we propose to adapt metamorphic testing to the security testing of web-oriented software systems.

In this dissertation, we propose two test automation approaches. The first one can automate any attack that can be described through a set of attacker-system interactions, but it requires the manual specification of inputs and oracles. The second one fully automates the testing process, but cannot automate some types of attacks. Therefore, we investigate the types of security vulnerabilities (hereafter referred to as *weaknesses*) that can be addressed by our two test automation approaches. To this end, we rely on the weaknesses reported in the common weakness enumeration (CWE) database [CWE, 2020s]. Moreover, we provide testability guidelines that enable effective test automation with our approaches.

1.2 Challenges

Our primary goal in this dissertation is to develop automated, requirements-based security testing techniques for web-oriented software systems. Our contributions range from methods for specifying security requirements to automating security tests and alleviating the oracle problem. To achieve this goal, we address the following challenges:

- The existing templates [Sindre and Opdahl, 2005, Opdahl and Sindre, 2009, Firesmith, 2003, McDermott and Fox, 1999], which support the specification of security requirements in use case-driven development of Web-oriented software systems, are not sufficiently precise and unambiguous to support automation. Moreover, the elicitation of threat scenarios is not structured and analyzable in terms of making control flow structures explicit and distinguishing types of scenarios for successful attacks.
- There is a lack of template support for specifying mitigation schemes that can be reused and adapted for various security threats. Existing approaches only supports specifying the flow of events mitigating each specific threat scenario [Sindre and Opdahl, 2001].
- Most security testing approaches focus on a particular vulnerability (e.g., code injection vulnerabilities [Tripp et al., 2013, Appelt et al., 2014] and buffer overflows [Haller et al., 2013, Ognawala et al., 2016]). Further, they only deal with simple inputs such as strings and files.
- Model-based approaches used to generate test cases are based on interaction protocol specifications [Veanes et al., 2005, Silva et al., 2008] and they can potentially generate test cases for such complex attack scenarios [Lebeau et al., 2013]. However, these approaches require detailed system models, which are seldom produced by engineers.
- There exist approaches used to generate functional test cases from natural language [Carvalho et al., 2014, Wang et al., 2015a]. Unfortunately, they only target functional testing but security vulnerability testing.
- Security testing suffers from the oracle problem [Barr et al., 2015, Pezze and Zhang, 2014]. In many situations where potential vulnerabilities are tested, a test oracle may not exist, or may be impractical to specify. Several proposed security testing approaches assume the availability of an implicit test oracle [Barr et al., 2015].

- Metamorphic testing has been proposed to alleviate the oracle problem [Chen et al., 1998]. It has been applied in many domains such as computer graphics [Mayer and Guderlei, 2006, Guderlei and Mayer, 2007, Just and Schweiggert, 2009, Kuo et al., 2011b], Web services [Chan et al., 2007b, Sun et al., 2011, Zhou et al., 2012], and embedded systems [Tse and Yau, 2004, Chan et al., 2007a, Kuo et al., 2011a, Jiang et al., 2013]. However, it has not yet been applied in the context of security testing, except for a few approaches targeting the functional testing of security components (e.g., code obfuscators [Chen et al., 2016]).

1.3 Research Contributions

In this dissertation, we address the preceding challenges with the following contributions:

1. **The Restricted Misuse Case Modeling (RMCM) approach, a use case-driven modeling method, is proposed to capture the security requirements of web-oriented software systems**. We rely on misuse case diagrams [Sindre and Opdahl, 2005] to model security requirements in terms of use cases. A misuse case diagram describes the relations between actors (including users and attackers), use cases (i.e., functional use cases), misuse cases (i.e., malicious use cases which harm the system) and security use cases (i.e., system features support mitigating misuse cases). To elicit security threats and threat scenarios in a structured and analyzable form, we adopt the Restricted Use Case Modeling method (RUCM) [Yue et al., 2013] to write use case specifications. RUCM is based on a template and restriction rules to support the specification of precise and analyzable use case specifications. Nevertheless, RUCM does not support specifying misuse case specifications. We extend RUCM with new rules to help specifying misuse case specifications and mitigation schemes. Furthermore, we define a set of NLP-based algorithms to automatically check the consistency between requirement specifications and diagrams, and the conformance between requirement specifications and the proposed template.
   This contribution has been published in a journal paper [Mai et al., 2018a], presented as a journal first paper at RE'18 [Maalej et al., 2018] and is discussed in Chapter 4.
2. **Misuse Case Programming (MCP), an approach that automatically generates security test cases from misuse case specifications**. This approach adopts the Natural Language Programming concepts [Ballard and Biermann, 1979, Pulido-Prieto and Juárez-Martínez, 2017] to support test case generation. MCP relies on natural language processing techniques (i.e., semantic role labeling) to extract the concepts (e.g., inputs and activities) appearing in requirements specifications and generates executable test cases by matching the extracted concepts to the members of a provided test driver API. Besides generating executable test cases, MCP automatically identifies input entities, their relationships (e.g., one-to-one or one-to-many), and test oracles.
   This contribution has been published in a conference paper [Mai et al., 2018b] and the implemented toolset has been presented in a tool demo paper [Mai et al., 2019]. It is discussed in Chapter 5.

3. **The Metamorphic Security Testing (MST) approach that alleviates the oracle problem in security testing**. It facilitates the specification of metamorphic relations that capture security properties of the system. These metamorphic relations are then used to automate testing and detect vulnerabilities. This contribution consists of a domain specific language used to specify metamorphic relations, an Eclipse-based editor supporting specifying metamorphic relations and transforming them to Java code, a crawler used to automatically collect source inputs, and a metamorphic testing framework running metamorphic relations against the system under test. Moreover, we provide a catalog of 22 system-agnostic metamorphic relations to automate security testing in Web systems. The approach has been applied to discover known vulnerabilities of the EDLAH2 system [MiC, 2017] and two open source systems (i.e. Jenkins [Eclipse Foundation, 2020] and Joomla [Joo, 2020]). Furthermore, MST has shown to help discover a new vulnerability (CVE-2020-2162 [Sto, 2020]) in Jenkins.
   This contribution has been published in a conference paper [Mai et al., 2020b] and in a tool demo paper [Mai et al., 2020a]. It is discussed in Chapter 6.
4. **Testability guidelines to support the adoption of the proposed approaches.** To characterize the weaknesses that can be identified by the proposed approaches and thus enable engineers to determine when the proposed approaches fit their needs, we analyze the weaknesses reported by MITRE[1] in the common weakness enumeration database [CWE, 2020s]. We identify the weaknesses that can and cannot be automatically tested by using MCP or MST. A list of applicability and inapplicability conditions have been identified. In addition, we investigate the testability factors (e.g., observability, controllability) which affect the applicability of our proposed test automation approaches on web-oriented software systems. Based on this, we propose guidelines for software designers and software developers to improve the testability of web-oriented software systems. These studies and guidelines are discussed in Chapter 7.

1.4 Dissertation Outline

Chapter 2 describes case study systems and introduces background concepts used in this dissertation such as modeling functional requirements, natural language programming, and metamorphic testing.

Chapter 3 discusses related work.

Chapter 4 describes our restricted misuse case modeling approach for eliciting structured and analyzable security requirements and mitigation schemes in terms of diagram and specifications.

Chapter 5 presents our natural language programming approach that automatically generates executable test cases from threat scenarios in natural language.

Chapter 6 introduces our metamorphic security testing approach that alleviates the oracle problems and improve test automation.

[1] The MITRE Corporation (abbreviated as MITRE) is an American not-for-profit organization that manages federally funded research and development centers [MITRE Corporation, 2018]. It provides a number of cybersecurity services.

Chapter 7 presents our investigation of the applicability of the proposed approaches for various security weaknesses and provides design and implementation guidelines to improve testability.

Chapter 8 summarizes the dissertation contributions and discusses perspectives on future work.

Chapter 2

Background

In this chapter, we present the case study systems and the background concepts that are used throughout this dissertation. This chapter is organized as follows. Section 2.1 presents the representative case study systems considered in this dissertation to evaluate the proposed approaches. Section 2.2 describes an approach for modeling functional requirements as use case specifications and presents the definition of security requirements with use case templates and its challenges. Sections 2.3 to 2.5 respectively introduce Natural Language Processing (NLP), Natural Language Programming (NLRP), and Metamorphic Testing (MT), backbone of the solutions presented in this dissertation.

2.1 Case study systems

The work presented in this dissertation has been partially motivated by the case study systems developed in the context of a European Union (EU) project in the healthcare domain named EDLAH2 [EDLAH2, 2017b]. The project brings academic institutions and software development companies together in a consortium to enhance the lifestyle of elderly people through a *gamification-based* approach. Gamification transforms activities that we are normally reluctant to do, e.g., exercising regularly, into a competition [Deterding et al., 2011]. The objective of the EDLAH2 project is to provide a set of gamification-based services on mobile devices that engage and challenge clients (elderly people) to improve their physical, mental, and social activities.

To achieve this objective, the EDLAH2 consortium developed a Web-oriented software system, i.e., a set of software components that can run on multiple types of systems and devices, which include mobile and wearable device applications (services). In EDLAH2, mobile applications are used to incentivize elderly people to perform intellectual activities (e.g., solving logic-based games including Sudoku), while the wearable device applications are used to track physical activities (e.g., tracking heartbeat and footsteps via a bracelet device). These applications collect data to be stored in a central data repository. The EDLAH2 website (i.e., *iCare* [MiC, 2017]) provides access to the data collected by the applications. It allows the carers of the clients to create user accounts and to configure the system (e.g., selecting mobile applications to install on the client's devices). The EDLAH2 system has thus a multi-tier and multi-device architecture, in which mobile applications interact with Web applications (i.e., software applications running on a Web server), Web applications interact with databases and third-party software, and mobile applications interact with mobile device data storage (e.g., SQLite or SD cards).

The objective of EDLAH2 entails that end-users (i.e., elderly people) provide access not only to their personal data but also to their daily activities. Hence, EDLAH2 is representative of contexts where engineers face the significant challenge of defining and ensuring security requirements in systems that process users' private data which is produced and shared by multiple components. Ensuring security in such contexts is complicated by multiple factors such as the presence of multiple components, communication over networks, and complex information flows involving multiple actors (e.g., end-users, mobile apps, third party software, and Web apps).

Web-oriented software systems like EDLAH2 are thus exposed to numerous security threats such as information disclosure, information modification, unauthorized access, and denial of service. A security threat becomes a reality in the presence of vulnerabilities that can be exploited by an attacker. Vulnerabilities might be introduced because of different reasons, ranging from the incomplete identification of security requirements to the adoption of bad programming practices, or the misconfiguration of software components and libraries. For example, disclosure of customer information may depend on improper requirements analysis, (e.g., the software analyst does not realize that the system should not expose the email addresses registered on the platform). SQL injection (SQLI) attacks often depend on bad programming practices (e.g., SQL statements created without relying on standard libraries that include sanitization mechanisms). Cross Site Scripting (XSS) vulnerabilities may depend on misconfiguration of default Javascript protection options. To systematically determine countermeasures which mitigate security threats, it is thus crucial to explicitly model both the activities that the system should perform to protect itself and the potential security threats that depend on the type of software components being developed.

In this dissertation, we use the Web-oriented software system developed for EDLAH2 as a case study to motivate, illustrate, and assess our proposed approaches. However, to ensure generalizability, we also apply the developed solutions to well-known Web-oriented systems, that is, Jenkins [Eclipse Foundation, 2020] and Joomla [Joo, 2020], which are illustrated below.

The second case study, Jenkins [Eclipse Foundation, 2020], is a free and open-source automation server. It is commonly used for building projects, running tests, static code analysis, and deploying, facilitating continuous integration and continuous delivery. We choose Jenkins as a case study since it is widely adopted by a large number of developers and well-known companies [app, 2020] such as ebay [eba, 2020], Facebook [Fac, 2020], LinkedIn [Lin, 2020], Netflix [Net, 2020], SpaceX [Spa, 2020]. Its Web interface includes advanced features such as Javascript-based login and AJAX interfaces thus beeing a representative case study for any modern Web-oriented system.

The third case study is the free and open-source content management system Joomla [Joo, 2020]. It is built on a model-view-controller Web application framework. Joomla supports multiple interfaces for end-users and administrators. It is one of the most popular content management systems. It is commonly used for building web sites for multiple categories such as corporate websites and small business websites, online magazines, E-commerce, online reservations, and Government websites. For example, it is used to manage the content of popular Web sites such as Nintendo [Nintendo, 2020] and Data2EU [Data2.eu, 2020].

The case studies considered in this dissertation are therefore different and provide complementary perspectives. EDLAH2 is developed in PHP [The PHP Group, 2017] and based on the Drupal content management system [Dru, 2017]; Jenkins is a Java Web application that can be executed within any servlet container [Eclipse Foundation, 2018]; Joomla is a PHP-based [The PHP Group, 2017] system.

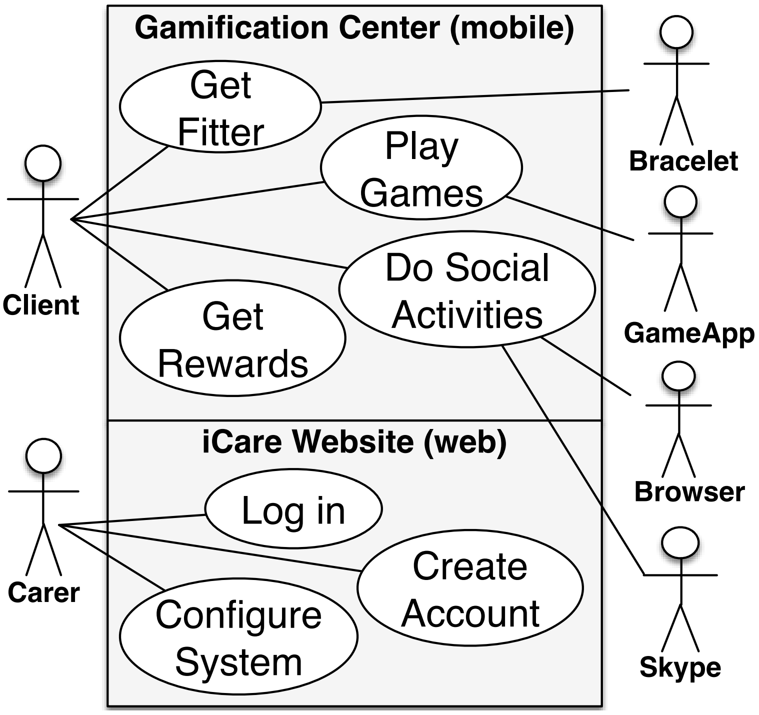


Figure 2.1. Part of the use case diagram of EDLAH2

2.2 Requirements Elicitation

In this dissertation, we define a methodology for modelling security requirements that integrates well with use case-driven solutions to capture functional requirements. Our motivation lies on the fact that, in addition to be one of the most popular agile solutions for capturing functional requirements [Larman, 2002], use case-driven solutions have been successfully adopted by methodologies for the automated generation of functional test cases [Wang et al., 2015a, Wang et al., 2015b, Wang et al., 2017, Wang et al., 2018, Wang et al., 2020].

Our reference use case-driven method for capturing functional requirements involves UML use case diagrams and use case specifications for describing functional requirements. This practice has been adopted in EDLAH2.

Fig. 2.1 depicts part of the use case diagram of EDLAH2. *Client* and *Carer* are the main actors of the system while *Bracelet*, *Game App*, *Browser* and *Skype* are the secondary actors representing the third-party apps. The use cases describe seven main functionalities: *get fitter*, *play games*, *do social activities*, *get rewards*, *log in*, *create account*, and *configure system.*

A use case specification contains a detailed description of a use case and usually conforms to a template [Cockburn, 2001, Armour and Miller, 2001, Kulak and Guiney, 2003, Yue et al., 2013]. The Cockburn template [Cockburn, 2001] is a possibility to specify functional use cases of EDLAH2. Fig. 2.2 shows two examples of such specifications that are part of EDLAH2. *Log in* describes how the carer logs into the system via the iCare website. *Get Fitter* describes how the client checks his

```
USE CASE Log in
Precondition The system displays the login screen.
Basic Path
1. The carer enters the user name and password in the login form.
2. The system checks in the browser if the user name and password are valid.
3. The system builds a database query using the user name and password.
4. The system evaluates the query in the database.
5. The system checks that the query is successful.
6. The system displays the welcome message.
Postcondition The carer has successfully logged in the system.
Alternative Paths
2a. The entered user name or password is invalid.
  2a1. The system displays the wrong user name or password message.
5a. The query is unsuccessful.
  5a1. The system displays the database error message.

USE CASE Get Fitter
Precondition The client has been successfully logged into the system.
Basic Path
1. The client requests to get current measurement.
2. The system receives the heart beat rate data from the bracelet.
3. The system checks whether the received heart beat rate data is correct.
4. The system stores the received data.
5. The system sets one point as a reward for the client.
6. The system displays the received heart beat rate data.
7. The system displays one point as a reward.
Postcondition The heart beat rate data and reward have been stored.
Alternative Paths
1a. The client requests to get activity data.
  1a1. The system receives the client's activity data from the bracelet.
  1a2. The system checks whether the activity data is correct.
  1a3. The system stores the activity data.
  1a4. The system sets two point as a reward for the client.
  1a5. The system displays the activity data.
  1a6. The system displays two points as a reward.
   1a3a. The system displays the error for incorrect activity data.
4a. The system displays the error for incorrect heart beat rate measurement.
```

Figure 2.2. Sample use case specifications for part of the EDLAH2 system

physical activities and condition (e.g., heart beat rate, number of steps and minutes of walking) as measured by the wearable device (i.e., bracelet).

Unfortunately, the Cockburn template does not support any explicit control flow structure. It is not feasible to specify a complex use case specification with loops by using Cockburn template. Moreover, it does not enable automatically test cases generation from use case specifications. We therefore chose a more structured and analyzable form of use case specifications, i.e., Restricted Use Case Modeling (RUCM) template [Yue et al., 2013].

Table 2.1. Restricted Use Case Modeling (RUCM) Template

Use Case Name	The name of the use case.	
Brief Description	Summarizes the use case in a short paragraph.	
Precondition	What should be true before the use case is executed.	
Primary Actor	The actor which initiates the use case.	
Secondary Actors	Other actors which interact with the system to accomplish the use case.	
Dependency	Include and extend relationships to other use cases.	
Generalization	Generalization relationships to other use cases.	
Basic Flow	Specifies the main successful path.	
	Steps(numbered)	Flow of events
	Postcondition	What should be true after the basic flow executes.
Specific Alternative Flow	Applies to one specific step of the basic flow.	
	RFS	A reference flow step number where flow branches from.
	Steps(numbered)	Flow of events
	Postcondition	What should be true after the alternative flow executes.
Global Alternative Flow	Applies to all the steps of the basic flow.	
	Steps(numbered)	Flow of events
	Postcondition	What should be true after the alternative flow executes.
Bounded Alternative Flow	Applies to more than one step of the basic flow, but not all of them.	
	RFS	A list of reference flow steps where flow branches from.
	Steps(numbered)	Flow of events
	Postcondition	What should be true after the alternative flow executes.

2.2.1 Restricted Use Case Modeling (RUCM)

RUCM [Yue et al., 2013] is a methodology for capturing use case specifications. It is based on a template and restriction rules, reducing the imprecision and incompleteness in use cases. We use RUCM to specify EDLAH2 system requirements including security requirements because it is designed to make use case specifications more precise and analyzable, while preserving their readability. The RUCM template consists of eleven fields, which are described in the Table 2.1. The structure of the RUCM template is specified by a set of restriction rules that are summarized in Table 2.2 and Table 2.3.

To reduce the ambiguity of use case specifications and to facilitate automated NL parsing, RUCM provides 16 restriction rules to constrain the use of natural language (named R1 to R16, see Table 2.2). RUCM includes additional 12 rules that specify the keywords to be used in specifications (i.,e. rules R17 to R28 in Table 2.3 except R26). Rules R27 and R28 had been introduced in follow-up work by Wang et al. [Wang et al., 2015a] to state additional keywords *SENDS ... TO* (see steps in Lines 9, 13, 18, 30-33, 39-42 in Fig. 2.3) and *REQUESTS ... FROM* (see steps in Lines 5, 6, 26, 28, 37 in Fig. 2.3) to specify data flow between the system and actors.

Fig. 2.3 shows the specifications in Fig. 2.2 modelled according to the RUCM template. The

Table 2.2. Restricted Rules (R1-R16)

#	Description	Explanation
R1	The subject of a sentence in basic and alternative flows should be the system or an actor.	Enforce describing flows of events correctly. These rules conform to our use case template (the five interactions).
R2	Describe the flow of events sequentially.	
R3	Actor-to-actor interactions are not allowed.	
R4	Describe one action per sentence. (Avoid compound predicates.)	Otherwise it is hard to decide the sequence of multiple actions in a sentence.
R5	Use present tense only.	Enforce describing what the system does, rather than what it will do or what it has done.
R6	Use active voice rather than passive voice.	Enforce explicitly showing the subject and/or object(s) of a sentence.
R7	Clearly describe the interaction between the system and actors without omitting its sender and receiver.	
R8	Use declarative sentence only. "Is the system idle?" is a non-declarative sentence.	Commonly required for writing use case specifications.
R9	Use words in a consistent way.	Keep one term to describe one thing.
R10	Do not use modal verbs (e.g., might).	Modal verbs and adverbs usually indicate uncertainty; Instead, metrics should be used if possible.
R11	Avoid adverbs (e.g., very).	
R12	Use simple sentences only. A simple sentence must contain only one subject and one predicate.	Facilitate automated natural language parsing and reduce ambiguity.
R13	Do not use negative adverb and adjective (e.g., hardly, never), but it is allowed to use not or no.	
R14	Do not use pronouns (e.g. he, this)	
R15	Do not use participle phrases as adverbial modifier. For example, the italic-font part of the sentence "ATM is idle, displaying a Welcome message", is a participle phrase.	
R16	Use "the system" to refer to the system under design consistently.	Keep one term to describe the system; therefore reduce ambiguity.

Table 2.3. Restricted Rules (R17-R28)

#	Description	#	Description
R17	INCLUDE USE CASE	R23	DO-UNTIL
R18	EXTENDED BY USE CASE	R24	ABORT
R19	RFS	R25	RESUME STEP
R20	IF-THEN-ELSE-ELSEIF-ENDIF	R26	Each basic flow and alternative flow should have its own postconditions.
R21	MEANWHILE	R27	SENDS-TO
R22	VALIDATE THAT	R28	REQUESTS-FROM

Precondition field describes the conditions that should hold to perform the use case. Line 2 in Fig. 2.3 indicates that the system should display the login screen before executing the Log in functionality. The *Primary Actor* field indicates the actor who initiates the use case (e.g., the carer is the primary actor of the use case Log in Fig. 2.3). The *Basic Flow* specifies the main successful path of the use case. It does not include any condition or branches. Each use case has only one *Basic Flow*. A *Basic Flow* consists of a sequence of steps and a postcondition. In Fig. 2.3, the *Basic Flow* of the use case *Log in* (Lines 4-10) consists of five steps, while the *Basic Flow* of the use case *Get Fitter* (Lines 25-34) consists of eight steps. *Alternative Flows* describe alternative sequences of steps. There are three types of Alternative Flows: Specific Alternative Flow, Bounded Alternative Flow, and Global alternative Flow [Yue et al., 2013]. The *Specific Alternative Flow* in Lines 11-15 in Fig. 2.3 (*Specific Alternative Flow* 1) describes an alternative scenario that takes place when entered user name is incorrect (i.e., when the condition in Line 3 is false). The keyword *RFS* (the restriction rule R19) is used in a specific (or bounded) alternative flow to refer to a step number (or a set of step numbers) of a reference flow (i.e., basic or alternative flows) that this alternative flow branches from. Line 12 in Fig. 2.3 specifies that the *Specific Alternative Flow* 1 of the use case *Log in* refers to step 3 of the *Basic*

1	**USE CASE** Log in
2	**Precondition** The system displays the login screen.
3	**Primary Actor** Carer
4	**Basic Flow**
5	1. The system REQUESTS the user name and password FROM the carer.
6	2. The system REQUESTS the password for the entered user name FROM the database.
7	3. The system VALIDATES THAT the entered user name is correct.
8	4. The system VALIDATES THAT the entered password is correct.
9	5. The system SENDS the welcome message TO the carer.
10	**Postcondition** The carer has been logged in the system.
11	**Specific Alternative Flow 1**
12	RFS 3
13	1. The system SENDS the invalid user name message TO the carer.
14	2. RESUME STEP 1.
15	**Postcondition** The carer has not been logged into the system. The system has displayed the invalid user name message.
16	**Specific Alternative Flow 2**
17	RFS 4
18	1. The system SENDS the invalid password message TO the carer.
19	2. RESUME STEP 1.
20	**Postcondition** The carer has not been logged into the system. The system has displayed the invalid password message.
21	
22	**USE CASE** Get Fitter
23	**Precondition** The client has been successfully logged into the system. The system displays the menu for the get fitter.
24	**Primary Actor** Client
25	**Basic Flow**
26	1. The system REQUESTS the choice FROM the client.
27	2. The system VALIDATES THAT the client's choice is getting current measurement.
28	3. The system REQUESTS the client's heart beat rate data FROM the sensor (bracelet).
29	4. The system VALIDATES THAT the client's heart beat rate data is correct.
30	5. The system SENDS the client's heart beat rate data TO the database.
31	6. The system SENDS one point reward TO the database.
32	7. The system SENDS one point reward to the client.
33	8. The system SENDS the client's heart beat rate data to the client.
34	**Postcondition** The client's heart beat rate data and reward (one point) have been stored in the database.
35	**Specific Alternative Flow 1**
36	RFS 2
37	1. The system REQUESTS the client's activity data FROM the sensor (bracelet).
38	2. The system VALIDATES THAT the client's activity data is correct.
39	3. The system SENDS the client's activity data TO the database.
40	4. The system SENDS two points reward TO the database.
41	5. The system SENDS two points reward to the client.
42	6. The system SENDS the client's activity data to the client.
43	**Postcondition** The client's activity data and reward (two points) have not been stored in the database.

Figure 2.3. Sample use case specifications in RUCM template for part of the EDLAH2 system

Flow. In other words, if the condition stated at the step 3 (Line 7) is not true, the *Specific Alternative Flow* 1 will be executed. The keyword *VALIDATES THAT* (the restriction rule R22) is used to state a condition checking (e.g., Line 7, 8 in Fig. 2.3). This means that the condition is validated by the system and must be true to proceed the next step. For instance, the condition in Line 7 has to be true to proceed the step 4 in Line 8. The keyword *RESUME STEP* (the restriction rule R25) is used when an alternative flow goes back to its corresponding basic/alternative flow. For instance, after executing all steps in the *Specific Alternative Flow* 1 (Line 14 in Fig. 2.3), the actor will go back to the step 1 of the *Basic Flow* (Line 5).

```
1    MISUSE CASE Get Unauthorized Access
2    Precondition At least one client account has already been created in the system.
3    Basic Path
4    1. The crook tampers with the values in the login URL.
5    2. The crook submits the tampered URL directly to the system.
6    3. The system builds a query using the values provided in login URL.
7    4. The system evaluates the query in the database.
8    5. The system checks that the query is successful.
9    6. The system displays the welcome message.
10   Postcondition The crook has gained some privileges.
11   Alternative Paths
12   5a. The query is unsuccessful.
13     5a1. The system displays the database error message, revealing some
             information about the database structure.
14     5a2. The crook tampers with the values in the login URL again
             based on the exposed information.
15     5a3. The crook submits the tampered URL directly to the system
             until the system checks that the query is successful.
16   5a3. The crook reaches maximum number of login attempts.
17     5a3a. The system displays the error message for login.
18   Mitigation Points
19   mp1. In Step 3, the system sanitizes the values before building the query.
20   mp2. In Step 5a1, the system does not replay the exact database error message
           and instead, it displays only non-confidential information.
21
22   MISUSE CASE Expose Information from Mobile
23   Precondition The mobile device also has a malware installed.
24   Basic Path
25   1. The malware requests access to user data stored in the system.
26   2. The system accepts the request.
27   3. The system sends user data to the malware.
28   Postcondition The malware has obtained user's private information.
29   Alternative Paths
30   2a. The system rejects the request.
```

Figure 2.4. Sample misuse case specifications for part of the EDLAH2 system

2.2.2 Elicitation of Security Requirements

Standard use case templates, such as Cockburn's, are insufficient to document security concerns in use case specifications [Sindre and Opdahl, 2001] [Sindre and Opdahl, 2005]. One state-of-the-art approach for eliciting security concerns, together with functional requirements, provides a misuse case specification template [Opdahl and Sindre, 2009] [Sindre and Opdahl, 2005] which extends a use case template with additional notions such as *misuse* and *mitigation point*. We applied this template in the context of the EDLAH2 project and attempted to elicit some of the security concerns for the use case specifications reported in Fig. 2.2. Some results of our attempt are shown in Fig. 2.4. The *Get Unauthorized Access* and *Expose Information from Mobile* misuse case specifications we targeted are similar to the example given in [Sindre and Opdahl, 2005]. The basic and alternative paths in Fig. 2.4 describe the sequence of actions that malicious actors go through to cause harm. The mitigation points document the actions in a path where the misuse case can be mitigated (Lines 19-20 in Fig. 2.4).

Based on this attempt, we identified three challenges that need to be considered when capturing security requirements in use case-driven development of Web-oriented software systems:

Eliciting security threats in an explicit, precise form (*Challenge 1*). We noticed that although the template we applied supports specifying various threats, it does not support their specification in a precise and unambiguous manner. This is the same for other related approaches such as [Sindre and Opdahl, 2005, Opdahl and Sindre, 2009, Firesmith, 2003, McDermott and Fox, 1999]. We identified three main limitations, reported in the following.

First, existing templates do not provide glossary or keywords for specifying common security threats. For instance, the *Get Unauthorized Access* misuse case in Fig. 2.4 corresponds to unauthorized access via SQLI (for categorizing the security threats, we follow the common, well-known terminology given in OWASP [OWASP, 2016]). In the specification, the term "SQL" was not even used. Likewise, the *Expose Information from Mobile* misuse case corresponds to information disclosure due to insecure data storage.

Second, existing templates do not provide a precise and systematic way to determine malicious actors in the specification. However, providing security extensions or keywords for precisely specifying common security threats, specific to the device type, would be useful. It would facilitate unambiguous communication among stakeholders and support various automated analyses, including security testing (e.g., identify the attacks that might hit a specific functionality and verify if its implementation properly prevents the attacks by simulating the attacker behaviour).

Third, existing templates do not explicitly distinguish between malicious actor-system interactions and other types of interactions. For instance, the steps in Lines 4-5 in Fig. 2.4 correspond to the malicious actor-system interactions, whereas the steps in Lines 6-9 correspond to the system's internal state changes. The interactions between malicious actors and the system contain information about the attack surfaces. But since the specification provided in Fig. 2.4 does not make this important difference, it might be difficult for a security tester to precisely determine where the attack surface is. In this case, the attack surface consists of the parameters in the login URL.

Eliciting threat scenarios in a structured and analyzable form (*Challenge 2*). The existing templates have two shortcomings in eliciting threat scenarios.

First, they do not have any explicit control flow structure. For instance, the *Get Unauthorized Access* misuse case in Fig. 2.4 tries a list of user name and password tuples iteratively until the malicious user logs into the system to get privileges. Since we do not have any explicit loop structure in the template we use for misuse cases in Fig. 2.4, we tried to describe the loop condition for the threat in non-restrictive natural language ('...until the system checks that...' in Line 15 in Fig. 2.4). However, it is not clear where the iteration starts in the execution flow.

Second, this does not allow to discern different types of scenarios — scenarios that a malicious actor may follow to successfully harm the system and scenarios that may not result in such harm. For instance, in the *Get Unauthorized Access* misuse case in Fig. 2.4, there are two alternative paths — the one starting from Line 12 leads to the scenario where the malicious actor harms the system and the other one starting from Line 16 leads to the scenario where the malicious actor fails to harm the system. As a result, it may not be easy for the stakeholders or an analysis tool to distinguish control flows and conditions leading to threat scenarios. Therefore, such specifications can be ambiguous and cannot support automated analyses.

Eliciting mitigation schemes (*Challenge 3*). After identifying security threats in threat scenarios, it is crucial to specify mitigation schemes matching these threats to demonstrate that the software design complies with applicable security standards and regulations. Such mitigation schemes provide the developers with guidance on how to prevent security threats specified in misuse cases. Different security threats and threat scenarios often share common mitigation methods and guidance. For instance, the two different security threats — information disclosure via SQLI and unauthorized access via SQLI — can both be mitigated by parameterizing the SQL queries. Existing work only supports specifying the flow of events mitigating each specific threat scenario (see Section 3.1). Such flows of events are embedded in misuse case specifications where one should specify the mitigation points (Lines 18-20 in Fig. 2.4). There is no structured way to specify the guidance for developers to mitigate security threats. In other words, there is a lack of template support for specifying mitigation schemes that can be reused and adapted for various security threats.

In this work, we aim to address these three challenges in a practical manner.

2.3 Natural Language Processing (NLP)

NLP techniques extract structured information from documents in Natural Language (NL) [Jurafsky and Martin, 2017]. They implement a pipeline that executes multiple analyses, e.g., tokenization, morphology analysis, syntax analysis, and semantic analysis. Each pipeline step produces results based on the output of the previous step.

In this dissertation, we rely on Semantic Role Labeling (SRL) [Jurafsky and Martin, 2017] as the NLP technique used to process requirements in NL. SRL is a semantic analysis technique that determines the roles played by the phrases in a sentence, e.g., the actor affected by an activity. For the sentences “The system starts” and “The system starts the database”, SRL determines that the actors affected by the actions are *the system* and *the database*, respectively. The component that is started coincides with the subject in the first sentence and with the object in the second sentence although the verb “to start” is used with an active voice in both. Therefore, this information cannot be captured by other NLP techniques like POS tagging and dependency parsing. The SRL roles can be effectively used to represent the meaning of a sentence in a structured form [Wang et al., 2018], which we need to generate API calls from a misuse case step.

To perform SRL, we rely on the CogComp NLP pipeline [University of Illinois, 2017] (hereafter CNP, which has shown to be effective in prior research work [Wang et al., 2018]. CNP tags the words in a sentence with keywords (e.g., *A0*, *A1*, *A2*, *AN*) to indicate the roles according to the PropBank model [Palmer et al., 2005]. *A0* indicates who (or what) performs an action, while *A1* indicates the actor most directly affected by the action. The other roles are verb-specific despite some

{The malicious user}$_{A0}$ {sends}$_{verb}$ {dictionary values}$_{A1}$ {to the system}$_{A2}$ {through the username and password fields}$_{AM-MNR}$

Figure 2.5. Example SRL tags generated by CNP.

Table 2.4. Some of the PropBank Additional Semantic Roles.

Identifier	Definition
AM-LOC	Indicates a location.
AM-MNR	Captures the manner in which an activity is performed.
AM-MOD	Indicates a modal verb.
AM-NEG	Indicates a negation, e.g. 'no'.
AM-PRD	Secondary predicate with additional information about A1.

commonalities (e.g., *A2* which is often used for the end state of an action). The PropBank model has also some other verb-independent roles (see Table 2.4). They are labeled with general keywords and match adjunct information in different sentences, e.g., *AM-NEG* indicating a negation.

Fig. 2.5 shows the SRL output for an example misuse case step. The phrase "The malicious user" represents the actor who performs the activity (tagged with *A0*); the phrase "dictionary values" is the actor affected by the verb (i.e., tagged with *A1*). The phrase "to the system" is the final location (tagged with *A2*), while the last chunk of the sentence represents the manner (tagged with *AM-MNR*) in which the activity is performed.

2.4 Natural Language Programming (NLRP)

Natural language programming (NLRP) refers to a family of approaches that automatically generate programs from NL specifications [Manning et al., 2014, Le et al., 2013, Guzzoni et al., 2007, Landhausser et al., 2017, Thummalapenta et al., 2012]. MCP has been inspired by techniques for building NL interpreters [Manning et al., 2014, Le et al., 2013, Guzzoni et al., 2007, Landhausser et al., 2017], in particular NLCI, a technique accepting action commands in English to translate them into executable code.

NLCI [Landhausser et al., 2017] translates NL sentences describing the system activities into sequences of API calls that implement the intended actions. To this end, it requires an ontology that captures the structure of the given API. To identify the API calls, NLCI relies on a scoring mechanism using the string similarity between the terms in the sentence and the method and parameter names in the API.

Although inspired by NLCI, our approach MCP differs from NCLI in many ways: MCP directly addresses security testing, MCP adopts a different scoring mechanism, and, finally, MCP supports the generation of assignment statements and (input) data structures (i.e., dictionaries), which is not supported by NLCI (see Section 5.6).

2.5 Metamorphic Testing (MT)

Metamorphic Testing is a technique used to alleviate the oracle problem. The core of MT is a set of Metamorphic Relations (MRs), which are necessary properties of the program under test in relation to multiple inputs and their expected outputs [Chen et al., 2018].

In MT, a single test case run requires multiple executions of the system under test with distinct inputs. The test outcome (pass or fail) results from the verification of the outputs of different executions against the MR.

As an example, let us consider an algorithm f that computes the shortest path for an indirect graph G. For any two nodes a and b in the graph G, it may not be practically feasible to generate all possible paths from a to b, and then check whether the output path is really the shortest path. However, a property of the shortest path algorithm is that the length of the shortest path will remain unchanged if the nodes a and b are swapped. Using this property, we can derive an MR, i.e.,

$$|f(G,a,b)| = |f(G,b,a)|$$

in which we need two executions of the function under test, one with (G,a,b) and another one with (G,b,a). The results of the two executions are verified against the relation. If there is a violation of the relation, then f is faulty.

We provide below basic definitions underpinning MT.

Definition 1 (Metamorphic Relation - MR). Let f be a function under test. A function f typically processes a set of arguments; we use the term *input* to refer to the set of arguments processed by the function under test. In our example, one possible input is (G,a,b). The function f produces an output. An MR is a condition that holds for any set of inputs $\langle x_1,...,x_n\rangle$ where $n \geq 2$, and their corresponding outputs $\langle f(x_1),...,f(x_n)\rangle$. MRs are typically expressed as implications.

In our example, the property of the target algorithm f is "the length of the shortest path will remain the same if the start and end nodes are swapped". The MR of this property is

$$(x_1 = (G,a,b)) \wedge (x_2 = (G,b,a)) \rightarrow |f(x_1)| = |f(x_2)|.$$

Definition 2 (Source Input and Follow-up Input). An MR implicitly defines how to generate a *follow-up input* from a *source input*. A source input is an input in the domain of f. A follow-up input is a different input that satisfies the properties expressed by the MR. In our example, (G,a,b) and (G,b,a) are the source and follow-up inputs, respectively.

Follow-up inputs can be defined by applying *transformation functions* to the source inputs. The use of *transformation functions* in MRs simplifies the identification of follow-up inputs. In our example, a transformation function that swaps the last two arguments of the source input can be used to define the follow-up input:

$$x_1 = (G,a,b) \wedge x_2 = \mathit{swapLastArguments}(x_1) \rightarrow |f(x_1)| = |f(x_2)|$$

where *swapLastArguments* is the transformation function.

Definition 3 (Metamorphic Testing - MT). MT consists of the following five steps:

1 Generate one source input (or more if required). In our example, a (random) graph G is generated; two vertices a and b in G are randomly selected for the source input.

2 Derive follow-up inputs based on the MR. In our example, the function *swapLastArguments* is applied to (G,a,b).

3 Execute the function under test with the source and follow-up inputs to obtain their respective outputs. In our example, the shortest path function is executed two times with (G,a,b) and (G,b,a).

4 Check whether the results violate the MR. If the MR is violated, then the function under test is faulty.

5 Restart from (1), up to a predefined number of iterations.

Chapter 3

Related Work

This chapter provides an overview of state-of-the-art solutions concerning the topics covered in this dissertation: modeling security requirements, automatically generating executable test cases, and addressing the oracle problem.

3.1 Modeling Security Requirements

There are numerous approaches in the literature to model security requirements [Turpe, 2017, Fabian et al., 2010, Mellado et al., 2010, Souag et al., 2016, Salini and Kanmani, 2012, Tondel et al., 2008, Anthonysamy et al., 2017, Beckers, 2012]. In a comprehensive literature review [Fabian et al., 2010], security requirements engineering methods were distinguished across six categories: *multilateral* (e.g., [Gurses et al., 2006, Gurses and Santen, 2006, Mead et al., 2005]), *UML-based* (e.g., [Sindre and Opdahl, 2005, Lodderstedt et al., 2002, Jürjens, 2003]), *goal-oriented* (e.g., [Liu et al., 2003, Elahi and Yu, 2007, Giorgini et al., 2005, van Lamsweerde, 2004, Mouratidis and Giorgini, 2007, Pasquale et al., 2016, Kalloniatis et al., 2008]), *problem frame-based* (e.g., [Lin et al., 2004, Hatebur et al., 2006, Hatebur et al., 2008, Haley et al., 2008, Haley et al., 2004, Thomas et al., 2014]), *risk/threat analysis-based* (e.g., [den Braber et al., 2007, Asnar et al., 2007, Cailliau and van Lamsweerde, 2013, van Lamsweerde, 2009, Asnar et al., 2011, Mayer et al., 2007]), and *common criteria-based approaches* (e.g., [CC, 2018, Mellado et al., 2006b, Mellado et al., 2006a]). The multilateral approaches follow the principles of multilateral security [Rannenberg et al., 1999] and focus on consolidating and reconciling the views of multiple stakeholders on the security requirements. Goal-oriented methods guide engineers towards the refinement of security [Liu et al., 2003, Elahi and Yu, 2007, Giorgini et al., 2005, van Lamsweerde, 2004, Mouratidis and Giorgini, 2007, Pasquale et al., 2016] features from high level requirements and have been applied to check whether a system meets its security requirements [Liu et al., 2003, Mouratidis and Giorgini, 2007], to identify trade-offs between security and other requirements [Pasquale et al., 2016], and to identify the architecture that suits the given goals [Kalloniatis et al., 2008]. The risk/threat analysis-based approaches consist of the identification of security goals and threats and focus on the analysis of the effects of these threats based on system specific information like trust relations among parties [Asnar et al., 2007], likelihood of incidents [den Braber et al., 2007], or data storage locations [Spiekermann and Cranor, 2008]. The problem frame-based approaches make use of the ideas underlying Jackson's problem frames [Jackson, 2001], and have been adopted to model both security [Lin et al., 2004, Hatebur et al., 2006, Hatebur et al., 2008, Haley et al., 2008, Haley et al., 2004] requirements. The common criteria-based approaches follow an international standard for information technology security evaluation [CC, 2018].

In the following, we discuss to what extent related work addresses the challenges identified in Section 2.2.2.

Eliciting security threats in an explicit, precise form. Most of the existing approaches rely on UML diagrams and use case specification templates to capture security threats. McDermott and Fox [McDermott and Fox, 1999, McDermott, 2001] propose *abuse cases* to describe harmful interactions (i.e., security threats) between a system and malicious actors, but relations between abuse cases and other types of requirements are not described. Sindre and Opdahl [Sindre and Opdahl, 2005] extend UML use case diagram with *misuse cases* and *security use cases* to model security threats (i.e., *misuse*), security-related requirements (i.e., *threat mitigation*) and other functional requirements. Alexander discusses automation to support misuse case diagrams [Alexander, 2003b, Alexander, 2003a] and reports experiences with misuse case diagrams in an industrial setting [Alexander, 2002]. Rosado et al. [Rosado et al., 2009] show how misuse case diagrams are employed to model the security requirements of a grid application, while Rostad [Rostad, 2006] extends misuse case diagrams with the notion of *vulnerability*, i.e., a weakness that may be exploited by attackers. Misuse case diagrams can be employed to represent misuse cases, security use cases and their relations, but not to capture security threats in misuse cases. To address this problem, Sindre and Opdahl [Sindre and Opdahl, 2001] adapt a use case specification template for detailed textual descriptions of threat scenarios. Common criteria-based approaches [Mellado et al., 2006b, Mellado et al., 2006a] apply the adapted template with misuse case diagrams [Sindre and Opdahl, 2005, Sindre and Opdahl, 2001] to elicit security threats and requirements. This template is extended for misuse case generalization [Sindre et al., 2002] and reuse [Sindre et al., 2003]. Deng et al. [Deng et al., 2011] employ the misuse case template [Sindre and Opdahl, 2001] in their privacy threat analysis framework to elicit privacy threat scenarios. Omoronyia et al. [Omoronyia et al., 2011] introduce two new fields into the template to highlight contextual properties of privacy misuse cases. Firesmith [Firesmith, 2003] proposes a similar template for *security use cases* which represent security-related requirements combined with a form of threat scenarios. However, these templates do not provide any construct or restriction rule to capture security threats in a precise and analyzable form to support automated analyses. El-Attar [El-Attar, 2012, El-Attar, 2014] proposes an approach to guide the analysts towards developing consistent misuse case diagrams and specifications. El-Attar's specifications can be automatically processed thanks to the presence of keywords, e.g., *misuse case* and *include*, that are used within the fields present in common use case templates. However, the keywords introduced by El-Attar do not support capturing security threats in an explicit form.

Goal-based approaches, such as KAOS [van Lamsweerde, 2004], Secure Tropos [Mouratidis and Giorgini, 2007], and attack trees [Swiderski, 2004], provide templates for specifying threats and common security goals. For instance, KAOS [van Lamsweerde, 2004] models threats as fault trees, also referred to as obstacle trees, in which the root is a negation (anti-goal) of a security goal. Based on the anti-goals, the approach determines how attackers may harm the system under design. PriS [Kalloniatis et al., 2008] is a goal-oriented methodology that focusses on privacy requirements. It includes a set of patterns that describe the processes to put in place in order to achieve a specific privacy goal, and provides a method to determine the architectural solution that fits the requirements. PriS mostly focusses on goals, and provides little support to model privacy threats. CORAS [den Braber et al., 2007], a risk analysis-based approach, rather focuses on the risks posed by security threats and on guiding the analysts in performing security risk analysis. It supports eliciting security threats using UML-like diagrams and some security-related notations at a high level. Rashid et al. [Rashid et al., 2016] employ the grounded theory method [Glaser and Strauss, 1967] and incident fault trees [Johnson, 2003] to discover and document emergent security threats which are implicit within and across a variety of security incidents. Risk-analysis approaches targeting privacy concerns instead focus on the definition of questionnaires that support analysts in the identification of privacy risks, but do not provide templates or models to capture the risks in a structured form [Hong et al., 2004, Spiekermann and Cranor, 2008].

Other approaches focus on security policy violations [Lodderstedt et al., 2002, Breaux et al., 2014, Breaux and Rao, 2013] and conflicting security objectives [Mead et al., 2005, Gurses et al., 2006]. SecureUML [Lodderstedt et al., 2002] is a modeling language for the model-driven development of secure, distributed systems based on UML, but its modeling support is limited to specifying policies for access control threats. Breaux et al. [Breaux et al., 2014, Breaux and Rao, 2013] propose a methodology which maps privacy requirements in natural language text to a formal language in description logic to detect conflicting privacy requirements causing threats. The analysis of conflicting privacy requirements is limited to the detection of conflicts between which data is actually processed and which data processing activity is declared in the policy. Multilateral approaches [Mead et al., 2005, Gurses et al., 2006] analyze general security needs of all the stakeholders of a system-to-be to consolidate different stakeholders' views on security threats. These approaches focus on identifying and resolving conflicts between different security goals of stakeholders. Their artifacts (e.g., UML models and attack trees) may explicitly capture interactions among security requirements as well as between security and functional requirements. Automated analysis of the artifacts to identify conflicts and ambiguities is also possible with some form of formalization work. However, the existing multilateral approaches provide rather conceptual frameworks. For example, SQUARE [Mead et al., 2005] only recommends to apply existing techniques that best suit the project at hand to model various threats.

In general, all the above-mentioned approaches provide systematic methods for modeling security threats and requirements in terms of security goals, anti-goals, attack trees, and/or access control policies at a high level. However, they do not focus on capturing security threats in a precise, detailed, and structured form. Therefore, automated analyses of the artifacts produced by these approaches are often not possible without incorporating additional formal notations such as temporal logics or logical formulas, which can be a tedious requirement even for developers and security analysts. In contrast to these approaches, our aim is to provide well-defined templates, restriction rules, and keywords to precisely capture various security threats in misuse case specifications. Our objective is to require knowledge on use case specification methods, thus enabling stakeholders to participate and communicate. Finally, we aim to propose templates, restriction rules and keywords that enable automated analyses (to identify conflicts and ambiguities) on the generated artifacts (i.e., misuse case diagrams and specifications).

Eliciting threat scenarios in a structured and analyzable form. The templates proposed for misuse cases [Sindre and Opdahl, 2001, El-Attar, 2012, El-Attar, 2014], security use cases [Firesmith, 2003] and abuse cases [McDermott, 2001] extend the common use case templates in the literature to elicit security requirements. But in general they do not provide any extension or any control flow structure to systematically identify and capture various threat scenarios. Some approaches employ UML models to use the control flow structures of UML in eliciting threat scenarios. For instance, Whittle et al. [Whittle et al., 2008] propose the use of sequence diagrams for the analysis of threat scenarios in misuse cases, while Sindre et al. [Sindre, 2007] incorporate malicious activities and malicious actors in UML activity diagrams to model potential attacks. Secure Tropos [Mouratidis and Giorgini, 2007] incorporates security rules in UML sequence diagrams. Song et al. [Song et al., 2005] propose to use aspect sequence diagrams to model access control-related security requirements. CORAS [den Braber et al., 2007] employs UML sequence and activity diagrams to model the behavior of the system under attack. UMLsec [Jürjens, 2001] combines several UML diagrams (e.g., statecharts and interaction diagrams) for modeling and analyzing threat scenarios. It also proposes some UML extensions (i.e., stereotypes, constraints, tagged values defined in a UML profile) to capture security concepts.

Attack trees [Schneier, 1999] and Microsoft's threat modeling approach [Swiderski, 2004], which can be considered as a special case of goal-based approaches, represent attacks or security threats against a system in a tree structure. The root node in the tree structure represents the attacker's goal while leaf nodes represent different ways of achieving that goal. A tree reflects the sequences of actions that must be carried out (a threat scenario) to realize a threat. However, it does not capture the conditions that must be met for the attack to happen. Problem frames are also used to model security threats [Lin et al., 2004, Lin et al., 2003, Hatebur et al., 2006]. They are basically diagrams representing the assets to be protected, the malicious subjects, the potential vulnerabilities of the system that the malicious subjects may exploit, and the environment through which the malicious users interact with the assets. Hatebur et al. [Hatebur et al., 2006] equip problem frames, representing threat scenarios, with preconditions and postconditions. Preconditions are specified using logical formulas to express what conditions must be met for a frame to be applicable.

Although the above-mentioned approaches provide diagrams or formalizations that employ a form of control flow structures in describing threat scenarios in a structured form, they do not provide any systematic way to distinguish scenarios that cause harm from those that do not. In addition, these approaches generally aim to aid the analysts for modeling threat scenarios using sequence/activity diagrams or formal methods. It is not clear whether stakeholders, who are not developers and with different levels of technical competence, can use these approaches.

In this dissertation, we propose an approach based on use cases, it allows different types of stakeholders to participate and communicate. We extend the RUCM template because it already provides control flow structures, e.g., *'do...while'* and *'if...then...else...'*, which can also be used for modeling threat scenarios. The new extensions we proposed capture success and failure scenarios in a structured and analyzable form.

Eliciting mitigation schemes. The template proposed by Sindre and Opdahl [Sindre and Opdahl, 2001] supports mitigation points where one can specify the flow of events mitigating each specific threat scenario. There is, however, no structured way to specify mitigation schemes, i.e., the guidance and methods for developers to mitigate security threats. For this reason, in this dissertation, we aim to provide template support for specifying mitigation schemes, which can be reused for mitigating various security threats. Further, to the best of our knowledge, the current approaches do not provide a way for stakeholders to demonstrate compliance of their software design against applicable security standards, which is an important requirement in many contexts. The mitigation and security use case templates proposed in this dissertation should enable traceability to demonstrate compliance with requirements.

3.2 Generating Test Cases

Security testing verifies the compliance of a software system with its security requirements [Felderer et al., 2016a, Felderer et al., 2016b, Hafner and Breu, 2008], which, in turn, can be given as *positive requirements*, specifying the expected functionality of a security mechanism, and as *negative requirements*, specifying what the system should not do [Meucci and Muller, 2014, Tian-yang et al., 2010]. For instance, a positive security requirement is “a user account is disabled after five unsuccessful login attempts”, while a negative security requirement is “a malicious customer should not be able to access resources that are dedicated to users with a different role (e.g., employee)”. This classification of security requirements is reflected in security testing [Meucci and Muller, 2014, Tian-yang et al., 2010, Felderer et al., 2016a]: (1) security functional testing validating whether the specified security properties are implemented correctly, and (2) security vulnerability testing addressing the identification of system vulnerabilities. Security vulnerability testing mimics attackers who aim to compromise the security properties of the system (e.g., confidentiality, integrity, and availability) [Felderer et al., 2016a, Arkin et al., 2005]. Security vulnerability testing requires specific expertise for simulating attacks (e.g., identifying risks in the system and generating tests driven by those risks), which makes test case generation and execution difficult to automate [Potter and McGraw, 2004].

Most vulnerability testing approaches focus on a particular vulnerability like buffer overflow and code injection vulnerabilities. For instance, Appelt et al. [Appelt et al., 2014] present an automated approach that generates test inputs for SQLi attacks, while Tripp et al. [Tripp et al., 2013] provide a learning algorithm for black-box detection of cross-site scripting (XSS) vulnerabilities. Ognawala et al. [Ognawala et al., 2016] present a tool that uses symbolic execution to detect memory out-of-bounds/buffer overflow vulnerabilities caused by unhandled memory operations in a program. Those approaches support engineers for a limited set of attack-related activities (e.g., input generation for an SQL injection vulnerability), and cannot be adopted to generate executable test cases for complex attack scenarios involving articulate interactions among parties, e.g., stealing an invitation email to register multiple fake users on a platform. For this reason, in this dissertation, we propose an approach, MCP, that enables engineers to specify such attack scenarios in misuse case specifications and automatically generates executable test cases from the specifications.

Model-based testing approaches are capable of generating test cases based on interaction protocol specifications [Veanes et al., 2005, Silva et al., 2008] and thus can potentially generate test cases for complex attack scenarios [Lebeau et al., 2013]. Model-based security testing is a relatively new research field [Felderer et al., 2016b], where some approaches have been proposed for security vulnerability testing (e.g., [Bertolino et al., 2012, Blome et al., 2013, He et al., 2008, Marback et al., 2013, Jürjens, 2008a, Masood et al., 2010a, Xu and Nygard, 2006, Martin and Xie, 2007b, Martin and Xie, 2007a, Martin et al., 2006]). For instance, Marback et al. [Marback et al., 2013] propose a model-based security testing approach that automatically generates security test sequences from threat trees. Wimmel and Jürjens [Wimmel and Jürjens, 2002] present an approach that generates security test sequences for vulnerabilities from a formal model supported by the CASE tool AutoFocus. Whittle et al. [Whittle et al., 2008] provide another approach that generates test sequences from attack scenarios in UML sequence and interaction overview diagrams. In these approaches, however, engineers have to transform the generated test sequences into executable tests manually, thus leading to limited benefits.

Xu et al. [Xu et al., 2012b, Xu et al., 2015] introduce the MISTA tool that automatically generates executable vulnerability test cases from formal threat models (i.e., Predicate/Transition - PrT nets). Jürjens [Jürjens, 2008a] relies on some security extensions of UML, i.e., UMLsec [Jürjens, 2002, Jürjens, 2005b, Jürjens, 2005a], to generate security vulnerability test cases from detailed UML statecharts capturing control and data-flow. Bertolino et al. [Bertolino et al., 2012] present a model-based approach for the automatic generation of test cases for security policies specified in a process algebra language. All these approaches require detailed formal models, which limits their adoption in industrial settings. In fact, engineers tend to avoid such detailed models because of the costs related to their development and maintenance, especially in contexts where system requirements are already available in NL.

There are approaches that generate functional system test cases from NL requirements (e.g., [Carvalho et al., 2014, de Figueiredo et al., 2006, Wang et al., 2015a, Wang et al., 2015b, Kaplan et al., 2008]). For instance, Wang et al. [Wang et al., 2015a, Wang et al., 2015b] automatically generate functional test cases from use case specifications written in RUCM. These approaches can be employed in the context of security functional testing, but not security vulnerability testing since they generate test cases only for the intended system behavior. Khamaiseh and Xu [Khamaiseh and Xu, 2017] present an approach that automatically builds, from misuse case specifications, PrT nets for the MISTA tool to automatically generate security vulnerability test cases. For the test code generation, test engineers have to provide some helper code and a model-implementation mapping description which maps the individual elements of a PrT net to their implementation constructs. In contrast, in this dissertation we propose test automation approaches that do not need any helper code or model-implementation mapping.

3.3 Addressing the Oracle Problem in Security Testing

Many vulnerability testing approaches rely on an implicit test oracle, i.e., one that relies on implicit knowledge to distinguish between correct and incorrect system behavior [Haley et al., 2008]. This is the case for approaches targeting buffer overflows, memory leaks, unhandled exceptions, and denial of service [Ognawala et al., 2016, Bekrar et al., 2011, Takanen et al., 2018], most of which rely on mutational fuzzing [Zeller et al., 2019], i.e., the generation of new inputs through the random modification of existing inputs. Implicit oracles deal with simple abnormal system behavior such as unexpected system termination and are not system-agnostic. What is abnormal in one system might be considered normal in a different context [Barr et al., 2015].

Vulnerability testing approaches for code injections also suffer from the oracle problem [Raghavan and Garcia-Molina, 2000, Kals et al., 2006, Martin and Lam, 2008, Bau et al., 2010, Appelt et al., 2014, Salas and Martins, 2014, Tripp et al., 2013, Appelt et al., 2013]. To resolve this problem, Huang et al. [Huang et al., 2003] proposed an MT-like technique which sends multiple HTTP requests, i.e., one request with an injection, an intentionally invalid request, and a valid request. They compare the responses to determine if the request with the injection is filtered. Unfortunately, MT-like approaches that address a broader set of security vulnerabilities are missing.

Model-based approaches [Felderer et al., 2016b, Felderer et al., 2011] typically target security vulnerability testing (e.g., [Bertolino et al., 2012, Blome et al., 2013, He et al., 2008, Marback et al., 2013, Jürjens, 2002, Jürjens, 2005b, Jürjens, 2005a, Jürjens, 2008b, Masood et al., 2010b, Xu and Nygard, 2006, Martin and Xie, 2007b, Martin and Xie, 2007a, Martin et al., 2006, Wimmel and Jürjens, 2002, Whittle et al., 2008, Xu et al., 2012b, Xu et al., 2015]) whereas a few solutions address security functional testing (e.g., [Le Traon et al., 2007, Mouelhi et al., 2008, Mouelhi et al., 2009, Xu et al., 2012a]). Most of these approaches only generate test sequences from security models and do not address the oracle problem. Approaches that generate test cases including oracles [Xu et al., 2012b, Xu et al., 2015, Xu et al., 2012a] rely on mappings between model-level abstractions (i.e., tokens in markings of PrT networks) and executable code implementing the oracle logic (e.g., searching for error messages in system output). Unfortunately, these approaches do not free engineers from implementation effort since they require the manual implementation of the executable oracle code. Furthermore, the model-based mapping supported by these approaches does not enable engineers to specify precise test oracles (e.g., oracles that verify the exact content of the output of the system with respect to its inputs [Xu et al., 2012b]).

With MT, we aim to address the limitations of security testing approaches. Indeed, MT supports oracle automation thanks to MRs that can precisely capture the relations between inputs and outputs. Considerable research has been devoted to developing MT approaches for various domains such as computer graphics (e.g., [Mayer and Guderlei, 2006, Guderlei and Mayer, 2007, Just and Schweiggert, 2009, Kuo et al., 2011b]), simulation (e.g., [Chen et al., 2009, Ding et al., 2011, Murphy et al., 2011]), Web services (e.g., [Chan et al., 2007b, Sun et al., 2011, Zhou et al., 2012]), embedded systems (e.g., [Tse and Yau, 2004, Chan et al., 2007a, Kuo et al., 2011a, Jiang et al., 2013]), compilers (e.g., [Tao et al., 2010, Le et al., 2014]), and machine learning (e.g., [Xie et al., 2009, Murphy et al., 2008]). Preliminary applications of MT to security testing [Chen et al., 2016] focus on the functional testing of security components (i.e., verifying the output of code obfuscators and the rendering of login interfaces) and the verification of low level properties broken by specific security bugs (e.g., heartbleed [Synopsys Inc., 2018]). Although these works show the feasibility of MT for security, they focus on a narrow set of vulnerabilities and do not automate the generation of executable metamorphic test cases, which are manually implemented based on the identified MRs.

Although MT is highly automatable, very few approaches provide proper tool support enabling engineers to write system-level MRs [Segura et al., 2016]. They require that MRs be defined either as Java methods [Zhu, 2015] or pre-/post-conditions [Murphy et al., 2009], which limit the adoption of MT to verify system-level, security properties. Furthermore, since MRs are often specified by capturing properties using a declarative notation, the use of an imperative language to implement the relations may force engineers to invest additional effort to translate abstract, declarative MRs.

To summarize, existing automated security testing approaches lack support for the generation of test oracles. The few approaches addressing the oracle problem either focus on a limited set of security vulnerabilities, or integrate oracles with limited capabilities. MT can overcome these limitations. It can be applied to both security functional testing and vulnerability testing since MRs can capture both security properties (e.g., a login screen should always be shown after a session timeout) and properties of the inputs and outputs involved in the discovery of a vulnerability (e.g., admin pages are accessed without authentication). Existing MT solutions target few, specific security bugs and do not support automated MT based on MRs capturing general security properties. To overcome these limitations, we need a DSL for MRs and algorithms that automate the execution of MT.

Chapter 4

Restricted Misuse Case Modeling

In this chapter, we propose, apply, and assess a modeling method that supports the specification of security requirements in a structured and analyzable form. Our motivation is that, in many contexts, use cases are common practice for the elicitation of functional requirements and should also be adapted for describing security requirements. We integrate an existing approach for modeling security requirements in terms of security threats, their mitigations, and their relations to use cases in a misuse case diagram. We introduce new security-related templates, i.e., a mitigation template and a misuse case template for specifying mitigation schemes and misuse case specifications in a structured and analyzable manner. Natural language processing can then be used to automatically report inconsistencies among artifacts and between the templates and specifications. We successfully applied our approach to an industrial healthcare project and report lessons learned and results from structured interviews with engineers. Since our approach supports the precise specification and analysis of security threats, threat scenarios and their mitigations, it also supports decision making and the analysis of compliance to standards.

4.1 Introduction

Modern internet-based services like home-banking [USC Credit Union, 2017], music-streaming [Spotify, 2017], food-delivery [DeliveryHero, 2017], and personal-training [FitBit, 2017] are delivered through Web-oriented software systems, i.e., software systems with components and interfaces that are executed on different types of devices including Web browsers, desktop applications, mobile applications, smart-TVs, and wearable devices. Most of the Web-oriented software systems process private end-user data collected and stored by different devices, such as credit balance reported by banking applications, locations visited by end-users, and health status tracked by personal training applications. The adoption of multiple devices augments security risks because of the presence of multiple attack surfaces (points at which security attacks can be executed), including malware that steals consumer and corporate data from smartphones [Jain and Shanbhag, 2012] and Web applications that unintentionally expose confidential data [Bortz and Boneh, 2007]. Therefore, security has become a crucial concern in the development of software systems, in this chapter we address the problem of facilitating the analysis of security requirements.

To identify the security requirements of a Web-oriented software system, it is necessary to take into consideration the characteristics of the specific service being developed and of the device types on which the service is going to be deployed. An example requirement of a home-banking smartphone service is that the user should automatically log off when the phone screen is locked to prevent phone thieves from accessing the bank account. This requirement is inappropriate for other types of services, e.g., personal training services which are used by runners and thus should be accessible without logging in, even after a screen lock (which normally happens while running). Examples of device specific characteristics that impact on security requirements include Web applications running on dedicated servers that are always online and thus prone to brute force attacks via the network. Mobile applications, instead, are often idle or offline, but they usually run on a device that is potentially shared with malicious applications inadvertently installed by end-users. Such applications can steal private data if it is not properly protected (e.g., through encryption). Therefore, it is crucial to precisely model and analyze security requirements of such Web-oriented software systems early in their development.

In this chapter, we propose, apply, and assess a use case-driven modeling method that supports the specification of security requirements of Web-oriented software systems in a structured and analyzable form. Use cases are one of the most common means adopted by software engineers to elicit requirements because they ease the communication between stakeholders [Larman, 2002]. Therefore, to achieve widespread applicability, the need for integrating security requirements with use case modeling warrants the development of a use case-driven, security requirements modeling method that is, in our context, tailored to the development of Web-oriented software systems.

Considerable research has been devoted to eliciting and analyzing security requirements using various forms of use cases (e.g., abuse cases [McDermott and Fox, 1999, McDermott, 2001], security use cases [Firesmith, 2003], and misuse cases [Opdahl and Sindre, 2009, Rostad, 2006, Sindre and Opdahl, 2005, Sindre and Opdahl, 2001]). However, as we detailed in Section 2.2.2, the applicability of these approaches in the context of security requirements modeling for Web-oriented software systems shows limitations with respect to (1) their support for explicitly specifying various types of security threats (a security threat is a possible event that exploits a vulnerability of the system to cause harm), (2) the definition of threat scenarios (a threat scenario is a flow of events containing interactions between a malicious actor and system to cause harm), and (3) the specification of mitigation for these threats.

These three features are essential in the type of business context we target where it is required to explicitly identify the threat scenarios that may affect important business operations in order to identify appropriate mitigation schemes and trade-offs between functional requirements and security concerns. It is also expected that such security requirements, specified in a structured and analyzable form, provide support for security testing, for example by helping with the identification of attack surfaces. In addition to specifying security threats, a common practice in many environments requires mitigation schemes to be documented for the stakeholders to demonstrate compliance with applicable security standards and regulations. However, existing approaches lack reusable templates to specify such mitigation schemes.

The goal of this chapter is to address the above challenges by proposing a use case-driven, security requirements modeling method called *Restricted Misuse Case Modeling (RMCM)*, which adapts existing methods and extends them. In RMCM, we employ misuse case diagrams proposed by Sindre and Opdahl [Sindre and Opdahl, 2005] to model security requirements in terms of use cases. Misuse cases describe attacks that may compromise use cases; security use cases specify how to mitigate such attacks. For eliciting security threats and threat scenarios in a structured and analyzable form, we adopt the Restricted Use Case Modeling method (RUCM) proposed in [Yue et al., 2013] to write use case specifications. RUCM is based on a template and restriction rules, reducing ambiguities and incompleteness in use cases. It was previously evaluated through controlled experiments and has shown to be usable and beneficial with respect to making use cases less ambiguous and more amenable to precise analysis and design [Wang et al., 2015a, Wang et al., 2015b, Hajri et al., 2015, Hajri et al., 2018b, Hajri et al., 2016, Hajri et al., 2017b, Hajri et al., 2017a, Hajri et al., 2018a]. However, since RUCM was not originally designed for modeling security requirements, we extend the RUCM template with new restriction rules and constructs, targeting the precise modeling of security threats. Further, we provide a template for mitigation schemes and three mitigation schemes that are pre-specified with standard and secure coding methods for mitigating common security threats. They can be readily used and revised as necessary.

RMCM employs Natural Language Processing (NLP) to report inconsistencies between a misuse case diagram and its RMCM specifications, and to analyze the compliance of such specifications against the provided RMCM templates. NLP is also used to identify and highlight the control flow leading to different threat scenarios and the steps in RMCM specifications that refer to interactions between malicious actors and the system. The latter provides security testers with information about attack surfaces on which security testing should focus. To summarize, the contributions of this chapter are:

- RMCM, a security requirements modeling method supporting the precise and analyzable specification of security threats, threat scenarios, and their mitigations, in the context of use case driven development of Web-oriented software systems;
- a practical toolchain, available at our tool website [Mai, 2017], including (1) a component that extends Papyrus [Papyrus, 2017] to support misuse case diagrams, (2) a component that extends IBM Doors [IBM Doors, 2017] to support misuse case specifications and mitigation schemes in the RMCM templates, and (3) a component relying on NLP to detect inconsistencies among these artifacts;
- a case study demonstrating the applicability of RMCM in a realistic development context involving multiple service and software providers in the healthcare domain.

This chapter is structured as follows. Section 4.2 provides an overview of RMCM. Section 4.3 focuses on the use case extensions in RMCM. In Section 4.4, we present our tool support. Section 4.5 reports on our industrial case study, from which we draw conclusions on the benefits and applicability of the proposed approach. Section 4.6 concludes this chapter.

4.2 Overview of the RMCM Modeling Method

The process in Fig. 4.1 presents an overview of our modeling method, *Restricted Misuse Case Modeling (RMCM)*. It is designed to address the challenges stated at the section 2.2.2 in the use case-driven development context we described for Web-oriented software systems, and builds upon and integrates existing work. The RMCM output is a misuse case diagram, use case specifications, security use case specifications, misuse case specifications, and mitigation schemes.

In Step 1, *Elicit requirements as use cases, security use cases and misuse cases*, the analyst elicits functional and security requirements relying on a misuse case diagram and the extended RUCM template (hereafter RMCM template), which are detailed in Section 4.3. Functional requirements and security requirements are captured in the misuse case diagram while it is further detailed in use case, security use case and misuse case specifications (*Challenges 1 and 2*). While use cases capture functional requirements, security use cases capture security countermeasures addressing potential attacks, which are themselves represented with misuse cases.

In Step 2, *Check conformance for diagram and specifications*, RMCM-V (Restricted Misuse Case Modeling - Verifier), the tool we developed for RMCM, automatically checks the consistency between the misuse case diagram and specifications, and between the specifications and the RMCM template. It relies on NLP. If there is any inconsistency, the analyst updates the diagram or specifications (Step 1). Steps 1 and 2 are iterative: the specifications and diagram are updated until the specifications conform to the RMCM template and they are consistent with the diagram.

In Step 3, *Elicit mitigation schemes for misuse cases*, mitigation schemes are elicited for the security threats specified in misuse cases (*Challenge 3*). Different from security use cases specifying the flow of events mitigating a specific threat scenario, mitigation schemes provide the secure coding methods adopted by the system, guidelines on how to educate users and other mechanisms to prevent various security threats in general. In Step 4, *Check conformance for mitigation schemes*, RMCM-V automatically checks whether the mitigation schemes conform to the mitigation template. Steps 3 and 4 are also iterative: the mitigation schemes are updated until they conform to the template.

The proposed method enables engineers to capture security threats and countermeasures. Risk analysis, i.e., ranking and prioritizing of security threats, is out of our scope. However, in contexts where risk analysis drives the engineering process (e.g., to prioritize test cases [Großmann and Seehusen, 2015]), the proposed approach can be integrated with existing risk analysis techniques, for

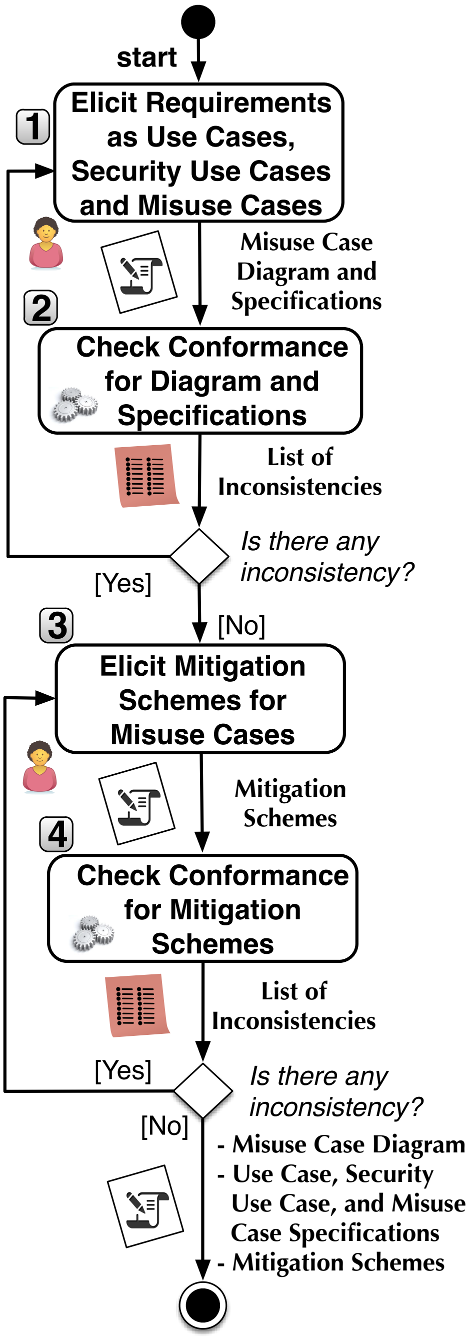

start
1
Elicit Requirements as Use Cases, Security Use Cases and Misuse Cases
Misuse Case Diagram and Specifications
2
Check Conformance for Diagram and Specifications
List of Inconsistencies
Is there any inconsistency?
[Yes]
[No]
3
Elicit Mitigation Schemes for Misuse Cases
Mitigation Schemes
4
Check Conformance for Mitigation Schemes
List of Inconsistencies
[Yes]
Is there any inconsistency?
[No]
- Misuse Case Diagram
- Use Case, Security Use Case, and Misuse Case Specifications
- Mitigation Schemes

Figure 4.1. Approach overview

example, techniques that rely on misuse case diagrams [Kim and Cha, 2012]. Similarly, the Common Vulnerability Scoring System (CVSS) [CVS, 2020] can be used to evaluate the risks due to vulnerabilities affecting the deployed production system.

In the following sections, we provide a detailed description of the steps of the proposed approach.

4.3 Capturing Security Requirements

In this section, we describe the artifacts produced by RMCM. We discuss how they were extended, compared to what was proposed in existing work, and illustrate how they address our three challenges with the running example.

4.3.1 Use Case Diagram with Misuse Case Extensions

To capture misuse cases, security use cases, use cases, and their relationships, RMCM relies on the misuse case extensions proposed by Sindre and Opdahl [Sindre and Opdahl, 2005] for use case diagrams. We made this choice for RMCM because of the explicit representation of misuse cases, security use cases, and their relationships (i.e., *threaten* and *mitigate*). In the following, we briefly introduce our extensions. The reader is referred to [Sindre and Opdahl, 2005] for further details. Fig. 4.2 depicts part of the misuse case diagram for EDLAH2.

As shown in Fig. 4.2, misuse cases, i.e., sequence of actions that a malicious actor can perform to cause harm, are greyed to distinguish them from use cases. Likewise, malicious actors (e.g., Malicious app) are distinguished from benign actors (e.g., Carer) and labeled with the keyword ‘malicious’. The UML stereotype «security» is used to distinguish security use cases that are countermeasures against misuse cases. In addition to the use case relationships (e.g., *include* and *extend*), *mitigate* is used for specifying the relationships between security use cases and misuse cases; and *threaten* is used for specifying the relationships between misuse cases and use cases [Sindre and Opdahl, 2005]. For instance, in Fig. 4.2, *Validate Website Inputs* mitigates *Get Unauthorized Access via SQLI*, which threatens *Log in*. *Expose Information via Insecure Data Storage* is an abstract misuse case that is extended by some concrete misuse cases threatening *Get Fitter*, *Play Games*, *Do Social Activities*, and *Get Rewards*.

4.3.2 Misuse Case and Security Use Case Specifications

Regarding misuse case specifications, to elicit security threats in a precise form and to elicit threat scenarios in a structured and analyzable form (*Challenges 1 and 2*), we propose the RMCM template, an extension of the RUCM template, shown in Table 4.1, and new restriction rules, shown in Table 4.2. The misuse case specifications are elicited using this template, further using the new restriction rules in addition to the original ones. These template and restriction rules are designed to make (mis)use case specifications explicit, precise, and analyzable by restricting the use of natural language and by

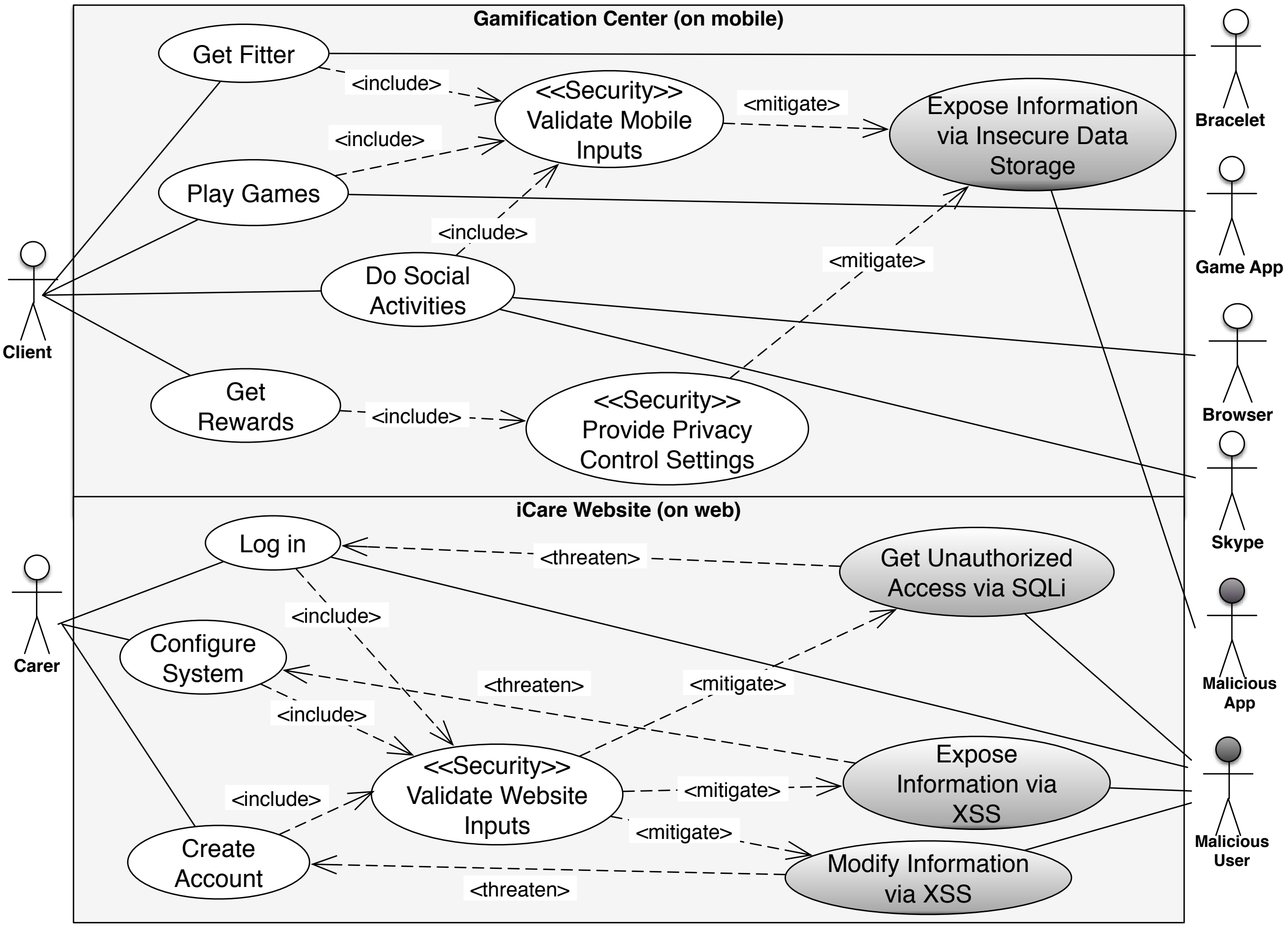


Figure 4.2. Part of the misuse use case diagram for EDLAH2

using specific keywords. Our extensions specifically target the modeling of security concerns for Web-oriented software systems.

Fig. 4.3 shows two simplified misuse case specifications of EDLAH2 written in RMCM, with all the RMCM keywords written in capital letters.

The original RUCM template provides basic and alternative flows which we adapted as *Basic Threat Flow*, *Specific/Bounded/Global Alternative Flow* and *Specific/Bounded/Global Alternative Threat Flow* (see Table 4.1). Threat flows specify unwanted incidents. Different from a basic flow in a use case specification, which describes a nominal scenario for an actor to use the system as intended, a basic threat flow describes a nominal scenario for a malicious actor to harm the system. It contains misuse case steps and a postcondition (Lines 6-13 and 38-41). A misuse case step can be one of the following interactions: a malicious actor initiates a security attack to the system (Lines 7, 8, 18, 19, 20 and 39); the system validates a request and/or data (Line 11); the system replies to a malicious actor with a result (Lines 12, 17, 27 and 40). A step can also capture the system altering its internal state (Lines 9 and 10). In addition, the inclusion of another use case can be specified as a step.

In RMCM, the assets impacted by a threat scenario are specified in the *Assets* field of the misuse case specifications. In addition, the assets should appear also in the postcondition of the basic

Table 4.1. Restricted misuse case modeling (RMCM) template

Misuse Case Name	The name of the misuse case.	
Brief Description	Summarizes the misuse case in a short paragraph.	
Precondition	What should be true before the misuse case is executed.	
Primary Actor	The actor which initiates the misuse case.	
Secondary Actors	The actors which interact with the system to accomplish the misuse case.	
Dependency	Include and extend relationships to other (mis)use cases.	
Generalization	Generalization relationships to other misuse cases.	
Threats	Threaten relationships to use cases.	
Assets	The assets (potentially) impacted by this threat.	
Basic Threat Flow	Specifies the main sequence of actions that the misuser carries out to harm the system.	
	Steps(numbered)	Flow of events
	Postcondition	The resulting unwanted condition and the asset(s) impacted after the threat flow executes.
Specific/Bounded/Global Alternative Threat Flow	A specific alternative sequence of actions that the misuser carries out to harm the system.	
	RFS	A reference flow step number where flow branches from.
	Steps(numbered)	Flow of events
	Postcondition	The resulting unwanted condition and the asset(s) impacted after the threat flow executes.
Specific/Bounded/Global Alternative Flow	A specific alternative sequence of actions that do not result in any harm to the system.	
	RFS	A reference flow step number where flow branches from.
	Steps(numbered)	Flow of events
	Postcondition	The resulting condition after the alternative flow executes.
Mitigation Scheme	Refers to the name of the mitigation scheme, specified using our mitigation template, to mitigate this misuse case. This complements security use case(s).	

threat flow or the specific/bounded/global alternative threat flows (Lines 13, 23, 30, 41 and 47). In RMCM, the purpose of postconditions is to capture the consequences that the activities of the malicious user/app have on the assets (Line 13).

In the following, we discuss with examples how the rules in Table 4.2 (*R1-R15*) are applied to address *Challenge 1* (see Section 2.2.2):

The step in Line 7 applies *R12* to explicitly specify the security threat in which a malicious actor, tagged with the 'MALICIOUS' keyword (*R1*), initiates an SQL injection attack through the two user input fields 'user name' and 'password' of the login URL. The 'SQLI' keyword (*R3*) is used to explicitly specify the type of security threat. The step in Line 8 specifies another threat in which the malicious actor bypasses the input validation method, possibly implemented on the client side (browser), and submits the login URL to the login server program directly (*R13*). Notice that in place of the 'SQLI' keyword as the value of the parameter ⟨attack⟩ in *R12*, the keywords 'XPATHI', 'XMLI', 'LDAPI', 'XSS', 'JSONI', 'BO', 'RCE' described in *R4-R10* can be used to explicitly elicit

```
1   MISUSE CASE Get Unauthorized Access via SQLi
2   Precondition At least one client account has already been created in the system.
3   Primary Actor MALICIOUS user
4   Threats Log In
5   Assets client DATA
6   Basic Threat Flow
7   1. The MALICIOUS user PROVIDES SQLI VALUES IN the user name and
      password fields of the login url.
8   2. The MALICIOUS user BYPASSES the login REQUEST TO the login
      server program.
9   3. The system builds a query with the values provided in the login url.
10  4. The system evaluates the query in the database.
11  5. The system VALIDATES THAT the query is successful.
12  6. The system SENDS the welcome message TO the MALICIOUS user.
13  Postcondition The MALICIOUS user accessed the client DATA without
                  authorization.
14  Specific Alternative Threat Flow
15  RFS 5
16  1. DO
17  2. The system SENDS the database error message DATA TO the MALICIOUS
      user.
18  3. The MALICIOUS user EXPLOITS the database error message DATA from
      the system.
19  4. The MALICIOUS user PROVIDES SQLI VALUES IN the user name and
      password fields of the login url.
20  5. The MALICIOUS user BYPASSES the login REQUEST TO the login
      server program.
21  6. UNTIL the query is successful.
22  7. RESUME STEP 6.
23  Postcondition The MALICIOUS user accessed the client DATA without
                  authorization.
24  Bounded Alternative Flow
25  RFS SATF1 1-6
26  1. IF the maximum number of login attempts is reached THEN
27  2. The system SENDS the invalid login message TO the MALICIOUS user.
28  3. ABORT.
29  4. ENDIF.
30  Postcondition The MALICIOUS user did not access the client DATA.
31  Mitigation Scheme Secure Coding for Server-side Program
32
33  MISUSE CASE Expose Information via Insecure Data Storage
34  Precondition The mobile device, in which the system is installed, also has a
              MALICIOUS app installed. The client has already used the system.
35  Primary Actor MALICIOUS app
36  Threats Get Fitter, Play Games, Do Social Activities
37  Assets user location DATA
38  Basic Threat Flow
39  1. The MALICIOUS app GETS the user location DATA FROM the log file
      of the system.
40  2. The system SENDS the user location DATA TO the MALICIOUS app.
41  Postcondition The MALICIOUS app obtained the user location DATA.
42  Specific Alternative Flow
43  RFS 2
44  1. IF the user location DATA is encrypted THEN
45  2. ABORT.
46  3. ENDIF.
47  Postcondition The MALICIOUS app did not obtain the user location DATA.
48  Mitigation Scheme Secure Coding for Mobile Program
```

Figure 4.3. Misuse case specifications in RMCM

Table 4.2. RMCM extensions

#	Description	Explanation
R1	MALICIOUS	Referring to a malicious actor to enforce explicitly describing the actions/steps that involve a malicious actor.
R2	DATA	Referring to the security-sensitive or privacy data to enforce explicitly describing the actions/steps that access or modify security-sensitive or privacy data.
R3	SQLI	Referring to SQL injection attacks to enforce explicitly describing the type of security threat.
R4	XPATHI	Referring to XPath injection attacks to enforce explicitly describing the type of security threat.
R5	XMLI	Referring to XML injection attacks to enforce explicitly describing the type of security threat.
R6	LDAPI	Referring to LDAP injection attacks to enforce explicitly describing the type of security threat.
R7	XSS	Referring to cross site scripting attacks to enforce explicitly describing the type of security threat.
R8	JSONI	Referring to JSON injection attacks to enforce explicitly describing the type of security threat.
R9	BO	Referring to Buffer overflow attacks to enforce explicitly describing the type of security threat.
R10	RCE	Referring to remote code execution attacks to enforce explicitly describing the type of security threat.
R11	GETS ⟨data⟩ FROM ⟨location⟩	Enforcing the explicit description of the security threats that leak data from the system (e.g., the MALICIOUS app GETS credit card DATA FROM log files).
R12	PROVIDES ⟨attack⟩ VALUES IN ⟨parameter⟩	Enforcing the explicit description of the security threat that exploits injection vulnerabilities. ⟨attack⟩ is the parameter in which the injection attack type (listed in R3-R10) is to be specified explicitly (e.g., the MALICIOUS user PROVIDES SQLI VALUES IN name and password).
R13	BYPASSES ⟨service-request⟩ REQUEST TO ⟨server-program⟩	Enforcing the explicit description of the security threats that enable a malicious actor to bypass any direct interaction with the client program and directly submit service requests to the server program (e.g., the MALICIOUS app BYPASSES view users REQUEST TO viewInfo program).
R14	EXPLOITS ⟨error-message⟩	Enforcing the explicit description of the security threat that exploits the information exposed in error or exception messages. The exposed information enables a malicious actor to understand the system better and conduct informed security attacks (e.g., the MALICIOUS user EXPLOITS exception message from the system).
R15	SENDS PRIVILEGED ⟨permission⟩ REQUEST TO ⟨client-program⟩	Enforcing the explicit description of the security threat that exploits insecure authorization schemes, which allows a malicious app to request the mobile app to execute privileged functionalities (e.g., the MALICIOUS app SENDS PRIVILEGED phone call REQUEST TO the main activity program).

other types of code injection security threats. The step in Line 18 applies *R14* to specify a security threat that exploits the information exposed in error or exception messages, tagged with the keyword 'DATA' (*R2*). The step in Line 39 applies *R11* to specify a security threat in which a malicious actor attempts to access the user location data, tagged with the keyword 'DATA' (*R2*), locally stored in the mobile device (specified by stating the location of the data: 'log file of the system' in the ⟨location⟩ parameter). Note that in the step in Line 12, the 'welcome message' is not tagged with the keyword 'DATA' because it is not security-sensitive and thus, not considered as an information asset.

Some of the RMCM extensions in Table 4.2 are based on the mobile security threats listed in well accepted standards such as CWE [CWE, 2020s] and OWASP [OWASP, 2016], and in more general threat modeling approaches such as STRIDE from Microsoft [Kohnfelder and Garg, 1999]. For instance, the security extension in R15 reflects permission re-delegation threats specific to mobile apps; also local storage problem of mobile apps is covered by the extension in R11.

In the following, we discuss with examples how *Challenge 2* (see Section 2.2.2) is addressed:

The 'VALIDATES THAT' keyword (Line 11), described in the original RUCM [Yue et al., 2013], indicates a condition that must be true to take the next step, otherwise an alternative flow is taken. It is one of the control flow structures we use for threat scenarios. In Fig. 4.3, the system proceeds to Step 6 (Line 12) if the query is successful (Line 11).

In the original RUCM template, there are three types of alternative flows: specific, bounded and global. In RMCM, we employ these alternative flows to describe failure scenarios for security attacks. A specific alternative flow always refers to and depends on a condition in a specific step of the basic threat flow. A bounded alternative flow refers to more than one step in the basic flow (Lines 24-30) while a global alternative flow refers to any step in the basic flow. For specific and bounded alternative flows, the keyword RFS is used to refer to one or more reference flow steps (Line 25).

In the RMCM template (Table 4.1), we introduce Specific/Bounded/Global alternative *threat* flows to describe alternative success scenarios and to distinguish them from failure scenarios for security attacks. For instance, in the *Get Unauthorized Access via SQLI* in Fig. 4.3, the specific alternative threat flow describes another success threat scenario (Lines 14-23) where the query is not validated by the system in the basic threat scenario (Line 11). The bounded alternative flow (Lines 24-30) describes the failure scenario for the attack given in this alternative threat flow (Lines 14-23).

Bounded and global alternative (threat) flows begin with the ‘IF ... THEN’ keyword, which is described in the original RUCM template, to describe the conditions under which alternative (threat) flows are taken (Line 26). Specific alternative flows do not necessarily begin with ‘IF .. THEN’ since a guard condition can be indicated in its reference flow step (Line 12). In addition, to describe threat scenarios, we also use other control flow structures — ‘DO ... UNTIL’ and ‘MEANWHILE’ — which are described in the original RUCM template. For instance, in the *Get Unauthorized Access via SQLI* misuse case in Fig. 4.3, the malicious user tries a list of user name and password tuples iteratively in an attempt to log in the system to obtain privileges. By having such explicit loop structure (Lines 16 and 21), we are able to specify where the iteration starts and ends in the execution flow of the threat scenario.

In RMCM, use case specifications are elicited according to the original RUCM template and restriction rules [Yue et al., 2013]. Following [Sindre and Opdahl, 2005], security use cases in RMCM specify the flow of events performed by the system to mitigate the attacks described in misuse cases. Differently from the original RUCM template, RMCM security use cases include two additional fields, ‘Compliance’ and ‘Mitigate’, to specify the standard provisions that the security use case should comply with and to specify the mitigated misuse case specifications (see Fig. 4.4).

SECURITY USE CASE Validate Website Inputs
Precondition The system has received some inputs.
Compliance ISO/IEC 27001:2013 clause A.9.4:System and application access control.
Mitigate Get Unauthorized Access via SQLi, Expose Information via XSS, Modify Information via XSS.
Basic Flow
1. The system sanitizes the inputs according to the input specification.
2. The system VALIDATES THAT the inputs are valid.
Postcondition The system has successfully validated the inputs.
Specific Alternative Flow
RFS 2
1. The system displays an error message.
2. ABORT.
Postcondition The system has shown the invalid characters in an error message.

Figure 4.4. A security use case specification

Table 4.3. Mitigation template

Scheme Name	The name of the mitigation scheme.	
Brief Description	A short description about the mitigation scheme.	
Actors	The actors who are responsible for reviewing and/or implementing the mitigation tasks.	
Mitigated Misuse Cases	Mitigate relationships to the misuse case(s). It specifies the misuse case(s) mitigated by the mitigation scheme.	
Compliance	Specifies the standard/applicable provision(s) that this mitigation scheme provides compliance.	
Mitigation Tasks	Specifies the mitigation tasks.	
	Tasks(numbered)	Mitigation Tasks

Fig. 4.4 shows a simplified security use case (only some fields are shown) for mitigating the threat *Get Unauthorized Access via SQLi*. It provides compliance (Line 3) with a clause in the widely-used security standard — ISO/IEC 27001:2013 Information Security Management Systems Requirements.

Even though the proposed templates are generic, our current security extensions (Table 4.2) focus on the threats specific to multi-device software ecosystems including mobile and desktop devices, as per the focus of our dissertation. For instance, RMCM has some mobile-specific extensions in Table 4.2 while other extensions (e.g., SQLI in R3 in Table 4.2) are specific to database-centric Web applications. However, our security requirements modeling method can a priori be adapted to other types of systems. The proposed RMCM and mitigation templates in Table 4.1 and Table 4.3 are generic enough to apply our security requirements modeling method to other application domains by introducing further security extensions into Table 4.2. However, since we have only evaluated our approach in the context of multi-device software ecosystems (see Section 4.5), we choose to remain conservative in our conclusions.

4.3.3 Mitigation Schemes

To address *Challenge 3* (see Section 2.2.2), RMCM provides the mitigation template given in Table 4.3. The field 'Mitigation Scheme' in misuse case specifications refers to the scheme mitigating misuse cases (Table 4.1). Mitigation schemes themselves are specified in a separate table, according to the mitigation template, to facilitate reuse. Differently from security use cases specifying flow of events mitigating a specific threat scenario, mitigation schemes provide the secure coding methods adopted by the system, guidelines on how to educate users and other mechanisms to prevent various security threats in general. As different security threats can be mitigated by applying standard secure coding methods, such as those listed in OWASP [OWASP, 2016], once a mitigation scheme is specified, it can be reused or tailored as necessary for various security threats. The field 'Mitigated Misuse Cases' in Table 4.3 lists such various security threats mitigated by a given scheme, while the field 'Compliance' lists the standard provisions that the mitigation scheme addresses.

Mitigation schemes have a different purpose than security use cases. While a security use case describes the sequence of activities that should be performed to implement an application specific requirement, mitigation schemes aim to be more general and capture compliance with standards, regulations and guidelines. Although, in general, mitigation schemes complement security use cases, in some situations these two specifications might be used to address the same security requirements. For example, data encryption, in addition to be a mitigation scheme item (see Task Item 2 of the mitigation scheme in Fig. 4.5), might be modelled as a security use case. The decision to model requirements with mitigation schemes or security use cases is taken by the software engineer based on the characteristics of the developed system, according to common practices. For example, a usual practice is to adopt use cases to model significant actor-system interactions but not to model a functionality exposed by third party services or software component interfaces (e.g., an API). According to this practice, data encryption is unlikely to be modelled as a security use case in systems that implement data encryption using standardized APIs.

Since mitigation schemes and security use cases are complementary, we have introduced the field 'Compliance' in both to provide precise traceability to specific clauses in standard provisions. Together, these artifacts provide a means for stakeholders to demonstrate compliance with applicable security standards and regulations. For instance, the mitigation scheme in Fig. 4.5 mitigates two misuse cases — *Expose Information via Insecure Data Storage* and *Expose Information due to Insecure Authentication*. It also supports compliance with some of the clauses in ISO/IEC 27001:2013. On our website [Mai, 2017], we give two additional mitigation schemes which are used to mitigate various security threats for EDLAH2.

Scheme Name	Secure Coding for Mobile Program.
Brief Description	This mitigation scheme mitigates serious and common security threats for mobile apps.
Actors	Software Developer, Security Engineer.
Mitigated Misuse Cases	Expose Information via Insecure Data Storage, Expose Information due to Insecure Authentication.
Compliance	ISO/IEC 27001:2013 clause A.6.1.5:Information security in project management, clause A.9.4:System & application access control, clause A.10.1:Cryptographic controls.
Mitigation Tasks	1 OBFUSCATE all apk files using an Android apk obfuscator. 2 ENCRYPT sensitive data stored in mobile device, such as SQLite database, cache and log files, and SD card. 3 Apply root detection check. If jailbreak is detected, the system warns the client of potential privacy data leakage. 4 Periodically clear caching data automatically. 5 Do not grant files world readable or writable permissions. 6 Perform code integrity violation check. 7 Educate users not to download apps from unofficial stores.

Figure 4.5. A sample mitigation scheme

One may argue that mitigation schemes seem to be no more than best secure coding practices with repetitions. We remark that the mitigation schemes precisely specify the actual practices adopted by the system (because not all of the security standards are applicable in practice for a specific system). Hence, they provide precise traceability to specific clauses in standard provisions for stakeholders to demonstrate compliance. In addition, mitigation schemes provide guidelines on how to educate users. As they are reusable for different security threats, repetitions would be minimal.

4.4 Tool Support

We have implemented a tool, *RMCM-V (Restricted Misuse Case Modeling-Verifier)*, for checking the consistency between the misuse case diagram and the specifications, and the compliance of the specifications with the RMCM template. RMCM-V reports inconsistencies such as a misuse case diagram missing a *threaten* or *mitigate* relationship in specifications. Section 4.4.1 describes the layered architecture of the tool. Section 4.4.2 presents the tool features with some screenshots. For more details and accessing the tool executables, see: `https://sntsvv.github.io/RMCM/`.

4.4.1 Tool Architecture

Fig. 4.6 shows the architecture of the tool. It consists of three layers: (i) *the User Interface (UI) layer*, (ii) *the Application layer*, and (iii) *the Data layer*.

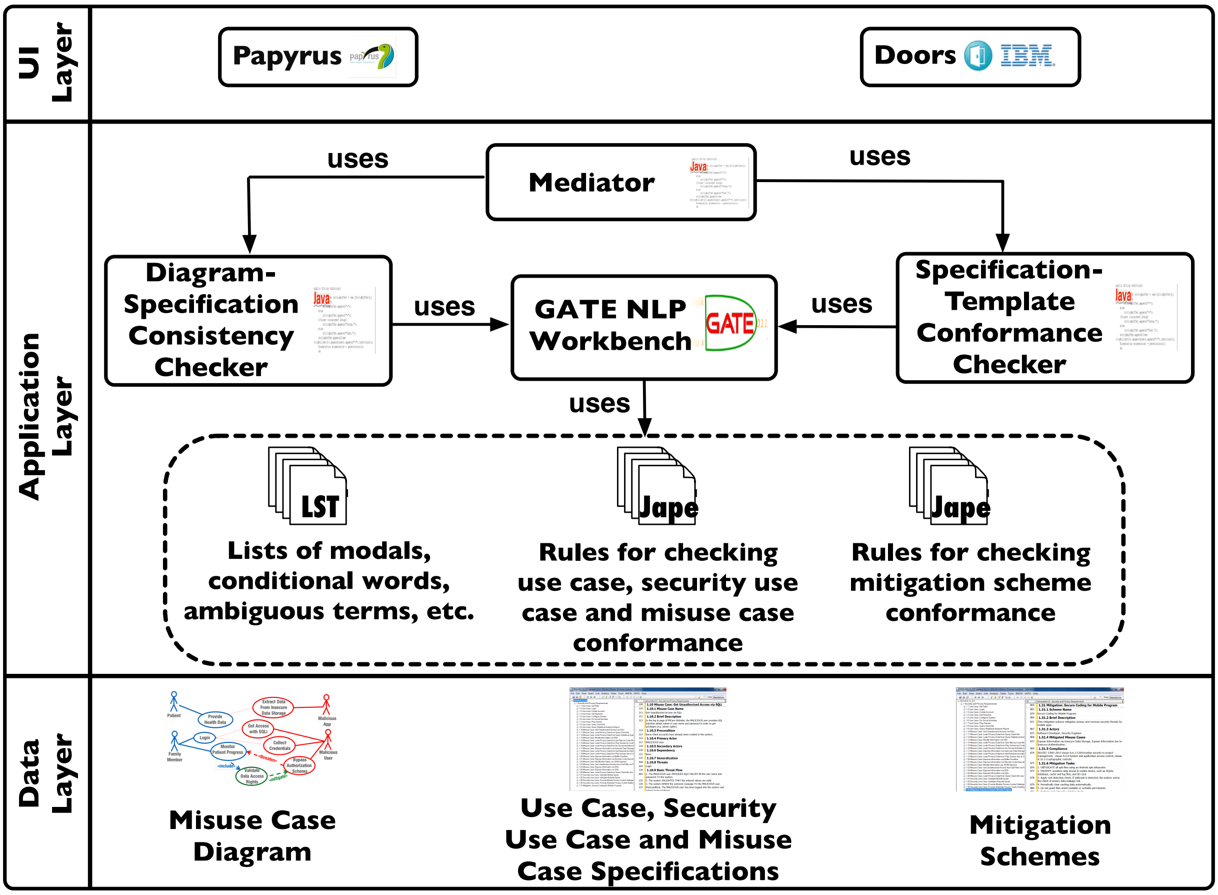

UI Layer
Papyrus
Doors
IBM
Application Layer
Mediator
uses
uses
Diagram-Specification Consistency Checker
uses
GATE NLP Workbench
uses
Specification-Template Conformance Checker
uses
LST
Lists of modals, conditional words, ambiguous terms, etc.
Jape
Rules for checking use case, security use case and misuse case conformance
Jape
Rules for checking mitigation scheme conformance
Data Layer
Misuse Case Diagram
Use Case, Security Use Case and Misuse Case Specifications
Mitigation Schemes

Figure 4.6. Layered architecture of RMCM-V

User Interface (UI) Layer. This layer supports the activities of eliciting security and (mis)use cases, and mitigation schemes (see Fig. 4.1). We employ IBM Doors [IBM Doors, 2017] for eliciting security and (mis)use case specifications and mitigation schemes according to the RUCM and RMCM templates and their restriction rules. We employ Papyrus [Papyrus, 2017] for misuse case diagrams. Sindre and Opdahl [Sindre and Opdahl, 2005] proposed a metamodel of the basic misuse case concepts and their relation to the UML metamodel. We adopted and implemented their proposed metamodel as a UML profile in Papyrus so that we can use the Papyrus model editor for drawing misuse case diagrams.

Application Layer. This layer supports the main activities of our modeling method in Fig. 4.1: *checking conformance for diagram and specifications* and *checking conformance for mitigation schemes*. It contains three main components implemented in Java: *Mediator*, *Diagram-Specification Consistency Checker*, and *Specification-Template Conformance Checker*. To access these *Application Layer* components through the *UI Layer*, we implemented an IBM Doors plugin.

The *Mediator* is a coordinator that manages the other two components. The *Specification-Template Conformance Checker* employs NLP to check whether the specifications and mitigation schemes comply with the RUCM and RMCM templates and their restriction rules. NLP is also used by the *Diagram-Specification Consistency Checker* to check the consistency between the misuse case diagram and specifications.

To support NLP, inspired by previous work in requirements engineering (i.e., [Arora et al., 2015a, Hajri et al., 2015, Hajri et al., 2018b, Arora et al., 2015b, Arora et al., 2015c]), we employ a regular expression engine, called JAPE [H. Cunningham et al, 2017], in the GATE workbench [GAT, 2017], an open-source NLP framework. We implemented the restriction rules in JAPE. First, the specifications are split into tokens. Second, Part-Of-Speech (POS) tags (i.e., *verb*, *noun*, and *pronoun*) are assigned to each token. By using the restriction rules implemented in JAPE, blocks of tokens are tagged to distinguish RUCM/RMCM steps (e.g., *actor to system interaction*, *malicious actor to system interaction*, and *internal actions*), types of flows (i.e., *threat-specific*, *alternative*, and *global*), and mitigation scheme tasks. The NLP output contains the annotated use case steps and mitigation scheme tasks. The *Diagram-Specification Consistency Checker* and *Specification-Template Conformance Checker* process these annotations with the misuse case diagram to generate the list of inconsistencies among artifacts.

Data Layer. The specifications and the mitigation schemes are stored as native IBM Doors format. The misuse case diagram is stored using the UML profile mechanism.

4.4.2 Tool Features

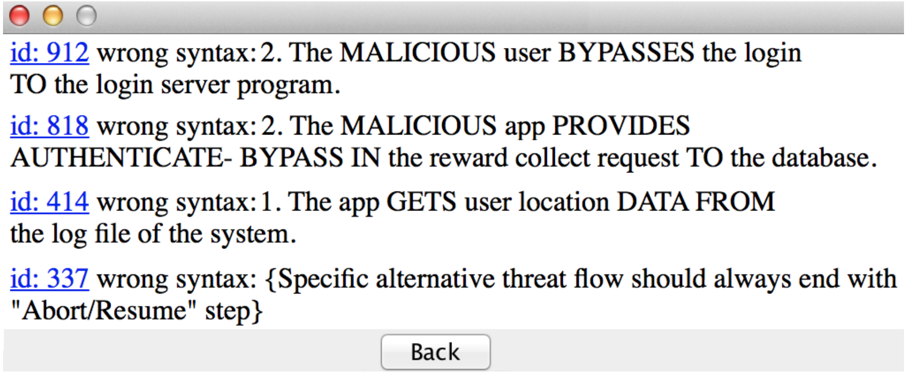
We describe the most important features of our tool: *managing RMCM artifacts*, *checking the conformance of RMCM specifications with the RMCM template*, *checking the consistency of the misuse case diagram and the RMCM specifications*, and *checking the conformance of mitigation schemes with the mitigation template*. These features support the steps of the modeling process given in Fig. 4.1.

Table 4.4. Some of the conformance checking rules for misuse case specifications

Modeling Element	Conformance Rules	
Misuse Case Steps in Threat Flows	1	A misuse case step in a threat flow should begin with a step head containing an ordinal number and a dot punctuation.
	2	Each misuse case step should contain a structure given in one of the RMCM extensions rules R11, R12, R13, R14 and R15 in Table 4.2 or a structure given in the original RUCM [Yue et al., 2013].
Specific Alternative Threat Flow	1	A specific alternative threat flow should have the header 'Specific Alternative Threat Flow'.
	2	A specific alternative threat flow should begin with the 'RFS' keyword which refers to a misuse case step in the basic threat flow or in another alternative threat flow.
	3	A specific alternative threat flow should have either an 'ABORT' step or a 'RESUME' step.
	4	A specific alternative threat flow should end with a post condition.
Bounded Alternative Threat Flow	1	A bounded alternative threat flow should have the header 'Bounded Alternative Threat Flow'.
	2	A bounded alternative threat flow should begin with the 'RFS' keyword which refers to a range of misuse case steps in the basic threat flow or in another alternative threat flow.
	3	The step after the 'RFS' step in a bounded alternative threat flow should include the 'IF...THEN' keyword.
	4	The last step of a bounded alternative threat flow should have the 'ENDIF' keyword.
	5	The step before the last step of a bounded alternative threat flow should be an 'ABORT' step or a 'RESUME' step.
	6	A bounded alternative threat flow should end with a post condition.
GETS ⟨data⟩ FROM ⟨location⟩	1	The subject of the sentence should start with the 'MALICIOUS' keyword followed by the 'GETS' and 'FROM' keywords, while any string can be used to represent ⟨data⟩.
PROVIDES ⟨attack⟩ VALUES IN ⟨parameter⟩	1	The subject of the sentence should start with 'MALICIOUS' keyword followed by the 'PROVIDES' and 'VALUE IN' keywords, while any string can be used to represent ⟨attack⟩ and ⟨parameter⟩.
BYPASSES ⟨req.⟩ REQUEST TO ⟨server-program⟩	1	The subject of the sentence should start with the 'MALICIOUS' keyword followed by the 'BYPASSES' and 'REQUEST TO' keywords, while any string can be used to represent ⟨req.⟩ and ⟨server-program⟩.

Managing RMCM artifacts. This feature supports Step 1, *Elicit Requirements as Use Cases, Security Use Cases and Misuse Cases*, and Step 3, *Elicit Mitigation Schemes for Misuse Cases*, in Fig. 4.1. The analyst can create, update, and delete the misuse case diagram, the corresponding specifications, and the mitigation schemes by using the selected modeling tools (i.e., IBM Doors and Papyrus) adopted in *RMCM-V*.

Checking the conformance of RMCM specifications with the RMCM template. The conformance of use case, misuse case and security use case specifications with the RMCM template and extensions needs to be ensured in Step 2, *Check Conformance for Diagram and Specifications*, in Fig. 4.1. Our tool automatically checks (1) if the use case and security use case specifications conform to the RUCM template [Yue et al., 2013] and (2) if the misuse case specifications conform to the RMCM template and the security extensions in Table 4.1 and in Table 4.2. Table 4.4 presents some of the conformance rules for misuse case specifications. For instance, a specific alternative threat flow should have the header 'Specific Alternative Threat Flow' followed by the 'RFS' keyword which refers to a misuse case step in the basic threat flow or in another alternative flow. To implement the conformance rules, RMCM-V leverages the information provided by the NLP framework; for example, in the case of the rule 'GETS ⟨data⟩ FROM ⟨location⟩', NLP enables RMCM-V to identify the noun phrase that corresponds with the subject of the sentence appearing in the use case step. NLP is required also to verify writing rules inherited from RUCM; for example, to foster clarity in requirements, RUCM requires that only the present tense be used and that adverbs be avoided. Both verb tenses and adverbs are determined using NLP.



id: 912 wrong syntax: 2. The MALICIOUS user BYPASSES the login TO the login server program.

id: 818 wrong syntax: 2. The MALICIOUS app PROVIDES AUTHENTICATE- BYPASS IN the reward collect request TO the database.

id: 414 wrong syntax: 1. The app GETS user location DATA FROM the log file of the system.

id: 337 wrong syntax: {Specific alternative threat flow should always end with "Abort/Resume" step}

Figure 4.7. A conformance checking result reported in RMCM-V user interface

Fig. 4.7 shows a sample result of the conformance checking of the use case and misuse case specifications for EDLAH2 in Section 4.3. Four types of inconsistencies are reported in Fig. 4.7: (i) the 'REQUEST' keyword of *R13* in Table 4.2 is missing in the misuse case specification, (ii) there is no 'AUTHENTICATE-BYPASS' keyword which can be used with the 'PROVIDES' keyword of *R12,* (iii) the 'MALICIOUS' keyword is missing before the 'GETS' and 'FROM' keywords of *R11*, and (iv) the 'ABORT/RESUME' step is missing in one of specific alternative threat flows in the misuse case specification. By clicking the links in the user interface, the user can access the non-conformant parts of the RMCM specifications (see 'id' links in Fig. 4.7).

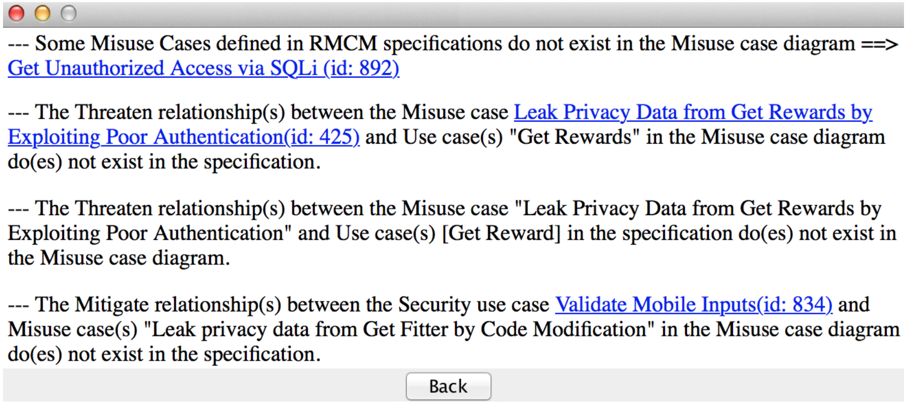
Checking the consistency of the misuse case diagram and the RMCM specifications. The consistency of the misuse case diagram and the corresponding specifications needs to be ensured as part of Step 2, *Check Conformance for Diagram and Specifications*, in Fig. 4.1. Table 4.5 presents some of the consistency rules for misuse case diagrams and RMCM specifications. For instance, for a misuse case threatening a use case in the misuse case diagram, the corresponding misuse case specification should have the threaten relation in its 'Threats' field (e.g., Lines 4 and 36 in Fig. 4.3). Fig. 4.8 presents an example output of the consistency checking of the misuse use case diagram and the corresponding specifications for EDLAH2 in Section 4.3.

Four types of inconsistencies are reported in Fig. 4.8: (i) a misuse case in the specifications does not exist in the misuse case diagram, (ii) the 'Threaten' relationship in the misuse case diagram does not exist in the specifications, (iii) the 'Threaten' relationship in the specifications does not exist in the misuse case diagram, and (iv) the 'Mitigate' relationship in the misuse case diagram does not exist in the specifications.

Checking the conformance of mitigation schemes with the mitigation template. The conformance of the mitigation schemes with the mitigation template needs to be ensured in Step 4, *Check*

Table 4.5. Some of the conformance checking rules for misuse case diagrams and specifications

Modeling Element		**Consistency Checking Rules**
Misuse Case Name	1	A misuse case specification should have a name.
	2	A misuse case in the misuse case diagram should have a name.
	3	Each misuse case specification name should match the name of the corresponding misuse case in the misuse case diagram.
	4	Each misuse case name in the misuse case diagram should match the name of the corresponding misuse case specification.
Associations between Misusers and Misuse Cases	1	A misuser should have a name with the 'MALICIOUS' keyword in the misuse case diagram and in the misuse case specifications.
	2	A misuser in the misuse case diagram should be described as *Primary Actor* or *Secondary Actor* in the corresponding misuse case specification.
	3	In the misuse case diagram, the relation between a misuse case and its misuser should be described using the association relationship.
	4	Each relation of a misuser and a misuse case in the misuse case specification should be given in the misuse case diagram, and vice versa.
The 'Threaten' Relationship	1	For a misuse case specification with the 'Threats' field referring to a use case specification, there should be a 'Threaten' relationship from the corresponding misuse case to the corresponding use case in the misuse case diagram.
	2	For a misuse case with a 'Threaten' relationship to a use case in the misuse case diagram, the corresponding missue case specification should have the 'Threats' field referring to the corresponding use case specification.
The 'Mitigate' Relationship	1	For a security use case specification with the 'Mitigate' field referring to a misuse case specification, there should be a 'Mitigate' relationship from the corresponding security use case to the corresponding misuse case in the misuse case diagram.
	2	For a security use case with a 'Mitigate' relationship to a misuse case in the misuse case diagram, the corresponding security use case specification should have the 'Mitigate' field referring to the corresponding misuse case specification.



--- Some Misuse Cases defined in RMCM specifications do not exist in the Misuse case diagram ==> Get Unauthorized Access via SQLi (id: 892)

--- The Threaten relationship(s) between the Misuse case Leak Privacy Data from Get Rewards by Exploiting Poor Authentication(id: 425) and Use case(s) "Get Rewards" in the Misuse case diagram do(es) not exist in the specification.

--- The Threaten relationship(s) between the Misuse case "Leak Privacy Data from Get Rewards by Exploiting Poor Authentication" and Use case(s) [Get Reward] in the specification do(es) not exist in the Misuse case diagram.

--- The Mitigate relationship(s) between the Security use case Validate Mobile Inputs(id: 834) and Misuse case(s) "Leak privacy data from Get Fitter by Code Modification" in the Misuse case diagram do(es) not exist in the specification.

Figure 4.8. RMCM-V user interface for reporting inconsistencies

Conformance for Mitigation Schemes, in Fig. 4.1. To do so, we derived some conformance checking rules from the mitigation template in Table 4.3. RMCM-V provides a conformance checking report identical to Fig. 4.7.

4.5 Evaluation

The goal of our evaluation is to assess, in an industrial context and on a case study, how our proposed modeling method RMCM and our tool RMCM-V can improve the practice of eliciting and analyzing security requirements, and how well they address the challenges that we identified in Section 2.2.2.

To this end, we first formulate four research questions:

- *RQ1:* Are the RMCM extensions to the RUCM template expressive enough to precisely and systematically model security threats?
- *RQ2:* Are the control flow structures of RMCM expressive enough to elicit the execution flow of threat scenarios in a structured form?
- *RQ3:* Does RMCM provide a structured way for stakeholders to specify guidance for mitigating common security threats?
- *RQ4:* Does RMCM-V provide useful automated assistance to correctly apply RMCM?

In light of the research questions given above, we evaluate our security requirements modeling method, RMCM, via reporting on (i) an industrial case study, i.e., EDLAH2, to demonstrate the feasibility of RMCM for a representative system (Section 4.5.1) and (ii) the results of a questionnaire survey along with discussions with EDLAH2 engineers, which aim at investigating how the approach is perceived to address the challenges listed in Section 2.2.2 and, furthermore, at gathering qualitative insights into the benefits and challenges of applying the method in an industrial setting (Section 4.5.2).

4.5.1 Industrial Case Study

We report our findings about the feasibility of our modeling method and its tool support in an industrial context. In order to experiment with RMCM in an industrial project, we applied it to the security requirements of the EDLAH2 project, which has been introduced in Section 2.1.

Table 4.6. The size of the RMCM artifacts in EDLAH2

	No.	**Relations**	**Alt. flows**	**Alt. threat flows**	**Steps**	**Malicious steps**
Use cases	9	15	26	NA	151	NA
Security use cases	4	28	7	NA	29	NA
Mitigation schemes	3	20	NA	NA	NA	NA
Misuse cases	17	20	25	9	216	26

To model the security requirements of EDLAH2 according to RMCM, we first examined initial EDLAH2 documentation consisting of a use case diagram and specifications, provided by the software engineers involved in the project and augmented with informal textual notes about security. Based on these artefacts, we worked together with EDLAH2 engineers to build and iteratively refine our models. EDLAH2 involves three different development teams for a total of ten software engineers. All the engineers working on EDLAH2 hold a master degree and some of them have more than ten years of software development experience. Since every team is responsible for different software components, the definition and refinement of the models has been performed independently by each

Table 4.7. Number of Occurrences of the RMCM Restrictions in EDLAH2

R1	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**	**R9**	**R10**	**R11**	**R12**	**R13**	**R14**	**R15**
142	120	4	0	0	0	9	2	2	2	2	15	1	6	2

team to mentor them to ensure that the methodology was applied properly. The training activity has been performed both during face-to-face project meetings and by sharing documents and tutorials in emails and online project meetings. Table 4.6 provides the size of the resulting RMCM artifacts.

In Table 4.6, the column ‘No.’ shows the numbers of use cases, security use cases, mitigation schemes, and misuse cases we modeled. The column ‘Relations’ shows the numbers of *include*, *mitigate*, and *threaten* relations among those artifacts. More precisely, in the case of the first row, the column ’Relations’ indicates the number of security use cases included by functional use cases, in the case of the second and third rows, the column ’Relations’ indicates the number of misuse cases mitigated by security use cases and mitigation schemes, while in the case of the fourth row the column ’Relations’ indicates the number of use cases threatened by misuse cases. The columns ‘Alt. flows’, ‘Alt. threat flows’, ‘Steps’, ‘Malicious steps’ show the numbers of alternative flows, alternative threat flows, steps, and malicious steps, respectively. ‘Malicious steps’ denotes the steps in misuse case specifications that correspond to interactions between malicious actors and the system. ‘NA’ denotes “not applicable”. In the following paragraphs, we rely on the data reported in Table 4.6 to respond to the research questions above.

RQ1

To support eliciting security threats in an explicit, precise form, RMCM includes two main extensions to RUCM, which are the identification of threaten relationships in misuse case specifications, and the adoption of specific keywords to capture common security threats.

To respond to *RQ1*, it is thus necessary to determine whether these modeling solutions (i.e., capturing threaten relationships and using security keywords) are useful, in practice, to precisely model security requirements. As an indirect measure of usefulness, we look at the number of occurrences of the threaten relationships and the security keywords in the misuse specifications of EDLAH2.

As shown in Table 4.6, we elicited 17 misuse cases threatening nine use cases, with a total of 20 threaten relationships among them. These numbers show that several threats tend to be relevant for each use case. This makes security requirements engineering rather complex, especially when involving many stakeholders, and it is therefore highly important to be systematic in identifying and specifying security requirements.

As for restriction rules, instead, we report in Table 4.7 the number of times each restriction rule (*R1-R15* in Table 4.2) is applied when eliciting misuse cases. As shown in Table 4.7, we applied almost all the proposed restriction rules to systematically model the security threats of EDLAH2. Only three restriction rules (*R4, R5, R6*) were not used since EDLAH2 uses an SQL database and they correspond to security threats targeting XML and LDAP databases.

Table 4.8. Number of Occurrences of the RMCM Control Flow Structures in EDLAH2

	Misuse Cases	Security Use Cases	Use Cases	Total
DO...UNTIL	10	0	0	10
IF...THEN	15	0	8	23
VALIDATES THAT	18	8	18	44
MEANWHILE	0	0	0	0
RESUME STEP	12	0	11	23
ABORT	21	8	15	44

Furthermore, it is interesting to note that some of the keywords introduced by RMCM to capture common security threats were not covered in the initial EDLAH2 documentation. For instance, the extensions helped us model that the misuse case *getting unauthorized access* can be performed by means of an SQLI injection attack (*R3* and *R12*) while the misuse case *exposing information from mobile* exploits insecure data storage (*R2* and *R11*), which was not previously documented. Capturing specific threats is useful since it helps engineers in identifying mitigation mechanisms to adopt.

Finally, RMCM keywords enable the precise identification of malicious steps, i.e., misuse case steps containing information about the attack surfaces. Typical attack surfaces include parameters, URLs, files, and programs. In total, we have identified 216 steps belonging to misuse cases, and among these, we have identified 26 malicious steps. The identification of malicious steps is important because it enables engineers to easily identify attack surfaces and the mechanisms to put in place in order to prevent these attacks.

RQ2

To respond to *RQ2*, we analyze the frequency of adoption of control flow structures in the misuse case specifications of EDLAH2. More precisely, we focus on the presence of alternative threat flows, which capture the conditions and the flow of events that may still lead to successful attacks when the attacks specified in the basic threat flows fail, and alternative flows, which capture the conditions under which a potential attack does not harm the system. In addition, we report also on the frequency of the control flow keywords appearing in the specifications.

Table 4.6 shows that we explicitly captured nine alternative threat flows (column 'alt. threat flows'), and 25 alternative flows (column 'alt. flows'), thus suggesting that a security threat can materialize through multiple threat scenarios which all need to be identified and carefully analyzed. It is therefore important to have a structured and precise mechanism to express such scenarios.

Table 4.8 reports the number of occurrences of the RMCM Control Flow Structures in the EDLAH2 specifications. As shown in Table 4.8, we made frequent use of all the RMCM control flow structures in misuse cases, except the 'MEANWHILE' structure for concurrency sentences. This happens because in EDLAH2 we did not have to model threat scenarios in which multiple activities are executed in parallel (e.g., because the presence of two malicious users is required to put in place a specific attack). More generally, misuse cases contain the same set of keywords appearing in use cases, with the exception of the keyword 'DO...UNTIL' which appears in misuse cases only. This

Table 4.9. Results from the analysis of non-conformant (mis)use case specifications in EDLAH2

Non-conformance Type	Explanation	Example
Unknown Step	A flow step does not follow the restricted rules for (mis)use case steps, i.e. actor-to-actor interactions, wrong keywords, wrong structures.	- RESUME step 3 (The STEP keyword should be uppercase) - The system PROVIDES CLIENT-SENSITIVE-INFO IN the parameters sent TO the game applications (The structure of the sentence should follow the rule R12 in the RMCM extensions)
Using Adverb in a Step	An adverb is used in a flow step. This violates the rule R11 in the original RUCM [Yue et al., 2013].	IF the user name in the entered account information is already in the system THEN (The adverb 'already' appears in the sentence).
More than One Action in a Step	There are two actions in a flow step sentence. This violates the rule R4 in the original RUCM [Yue et al., 2013].	The system REQUESTS the user name and password FROM the MALICIOUS user (This sentence should be split in two steps)
Wrong Structure of Specific Alternative Flows	A specific alternative threat flow uses RFS in a wrong way at the beginning of the flow, or the last step of the flow is not a valid Abort or Resume step.	- RFS SAF 2 (It should be written in the format of 'RFS SAF 1-2'. In this case, SAF 1 points out the first specific alternative flow) - There are three specific alternative threat flows using the wrong keyword 'RESUME STEP'. - There are two specific alternative flows ending without 'ABORT' or 'RESUME STEP'.

keyword is typically used to describe iterative attacks in misuse cases (e.g., malicious users trying to log into the system by trying a list of available usernames). In the use case specifications of EDLAH2, instead, we do not describe iterative behaviors of valid system users.

RQ3

To respond to *RQ3*, we focus on security use cases and mitigation schemes. Table 4.6 shows that we elicited four security use cases and three mitigation schemes. They typically mitigate more than one misuse case since there are 28 *mitigate* relations between security use cases and misuse cases and 20 *mitigate* relations between mitigation schemes and misuse cases (column 'relations'). These numbers show that both security use cases and mitigation schemes can be reused across multiple misuse cases and, therefore, they are useful and reusable artifacts that should be captured independently from the misuse cases.

RQ4

To respond to *RQ4*, we applied RMCM-V to check the conformance of the first version of the EDLAH2 misuse case specifications with the RMCM template, and to check the consistency of the misuse case diagram and the (mis)use case specifications in EDLAH2. RMCM-V reported 29 warnings matching four non-conformance types when analyzing the conformance of the EDLAH2 specifications with the RMCM template (see Table 4.9).

Two warnings among the twenty nine warnings in the conformance checking results (2 / 29 = 6%) were related to the wrong use of the security keywords in the RMCM extensions in Table 4.2. All other warnings were about the violation of the rules in the original RUCM template [Yue et al., 2013], e.g., more than one action being described in a single step.

Table 4.10. Results from the analysis of the inconsistent misuse case diagram and specifications in EDLAH2

Inconsistency Type	Explanation	Example
Lack of Security Use Cases	A security use case defined in the RMCM specifications does not exist in the misuse case diagram.	The Security Use Case 'Validate Mobile Inputs' is given in the specification, but it does not appear in the diagram.
Missing 'Include' Relations	Some 'Include' relations between use cases and security use cases in the misuse case diagram do not exist in the specifications.	Nine 'Include' relations in the misuse case diagram do not appear in the specifications. For instance, in the misuse case diagram, the use case 'Get Fitter' includes the security use case 'Provide Privacy Control Settings', but there is no relation between them in the specifications.
Missing 'Threaten' Relations	Some 'Threaten' relations between misuse cases and use cases in the misuse case diagram do not exist in the specifications, and vice versa.	The 'Threaten' relation between the misuse case 'Get Unauthorized Access via SQLi' and the use case 'Login' in the misuse case diagram does not exist in the specifications.
Missing 'Mitigate' Relations	Some 'Mitigate' relations between security use cases and misuse cases in the specifications do not exist in the misuse case diagram.	The 'Mitigate' relation between the security use case 'Provide Privacy Control Settings' and the misuse case 'Leak Privacy Data from Play Games due to Unintentional Data Flow' in the specifications does not exist in the misuse case diagram.
Missing Actor-(Mis)use Case Relations	Some Actor - (Mis)use case relations in the misuse case diagram do not exist in the specifications, and vice versa.	The relation between the use case 'Create Account' and the actor 'Manager' in the misuse case diagram does not exist in the corresponding specification.

RMCM-V reported 17 inconsistencies between the misuse case diagram and the (mis)use case specifications (see Table 4.10). Four of them (4 / 17 = 23%) are related to the misuse cases. All other inconsistencies are about missing use cases, missing 'Include' relations or missing actors in use cases.

We, together with EDLAH2 engineers, were able to correct, in one iteration, all the issues reported in Table 4.9 and in Table 4.10. Our main observation was that it was easy, after some training, for the EDLAH2 engineers to correctly use our security extensions. Additionally, we manually inspected the specifications and verified that RMCM-V was able to identify all the inconsistencies and parts of the misuse case specifications that did not conform with the RMCM template.

4.5.2 Questionnaire Study and Discussions with the Engineers

The questionnaire study is described and reported according to the template provided by Oppenheim [Oppenheim, 2005]. To qualitatively evaluate the RMCM output in light of the four research questions presented at the beginning of this section, we had semi-structured interviews with four engineers holding various roles in the EDLAH2 consortium (i.e., project manager, software engineer, and game architect). All participants have substantial software development experience, ranging from three to 28 years. All of them had experiences with use case driven development and modeling. The interview included a presentation illustrating the RMCM steps, a tool demo, and examples from EDLAH2. The participants of the questionnaire study were also involved in the case study reported in Section 4.5.1. To perform the case study, we, together with the participants, had multiple face-to-face project meetings. We had shared the documents and online tutorials with them. To confirm the misuse case models of EDLAH2, we had many technical meetings with the EDLAH2 engineers, including the participants of the questionnaire.

Table 4.11. Questionnaire for the Evaluation of RMCM with the average of the votes.

Question	Result
Misuse Case Diagrams	
1. The diagram is simple enough to enable communication between engineers and stakeholders.	2.50
2. If a misuse case diagram like the one we presented were available to you, would you use it to help you capture or understand security threats and mitigations?	3.00
3. The notation provides enough expressiveness to conveniently capture the security threats and mitigations in your projects	3.00
Misuse Case Specifications for Capturing Security Threats.	
4. Misuse case specifications are simple enough to enable communication between engineers and stakeholders.	2.75
5. If misuse case specifications like the ones we presented were available to you, would you use those specifications to help you capture or understand security threats?	2.75
6. Security threats captured in the misuse case diagram are adequately reflected in the specifications.	3.50
Mitigation Schemes for Capturing Secure Coding Methods.	
7. Mitigation schemes are simple enough to enable communication between analysts and programmers.	2.25
8. If mitigation schemes like the ones we presented were available to you, would you use those schemes to help you capture or understand secure coding methods for mitigating security threats?	2.50
Restricted Misuse Case Modeling Method for Security and Privacy	
9. The steps in our modeling method are easy to follow.	3.50
10. The effort required to learn how to apply our method is reasonable.	2.50
11. Would you see value in adopting the presented method for capturing security threats and mitigations?	2.75
12. Does the presented method provide useful assistance for easing the communication between engineers and stakeholders?	2.25
13. Does the presented method provide useful assistance for capturing and analyzing security threats compared to the current modeling practice in your projects?	2.50
14. Do you think that the presented tool provides useful assistance for minimising the inconsistencies in misuse case diagrams and specifications?	2.75

Score for answers to interrogative questions: 4 - *very probably*, 3 - *probably*, 2 - probably not, 1 - *surely not.* Score for answers to statements: 4 - *strongly agree*, 3 - *agree*, 2 - *disagree*, 1 - *strongly disagree*.

To capture the perception of the participants regarding the potential benefits of RMCM, and assess the extent to which it addresses the targeted challenges, we handed out a questionnaire [Edl, 2017] including questions to be answered according to a Likert scale [Oppenheim, 2005], along with open, written comments. The questionnaire was structured for the participants to assess RMCM in terms of adoption effort, expressiveness, and comparison with current practice. Table 4.11 shows the questions appearing in the questionnaire (divided by topic), along with the average of the scores for each answer (column result). The Likert scale answers provided in the questionnaire were '*strongly agree*', '*agree*', '*disagree*', and '*strongly disagree*' for statement sentences in the questionnaire (e.g., question one in Table 4.11) while '*very probably*', '*probably*', '*probably not*', and '*surely not*' were for interrogative sentences (e.g., question 2). A discussion of the questionnaire results in light of the four research questions driving the study follows.

RQ1

The answers given to the first six questions in Table 4.11 indicate that the RMCM extensions provide enough expressiveness to conveniently capture security requirements in EDLAH2. More precisely, the answers to questions one and four indicate that misuse case diagrams and misuse case specifications properly support the communication between stakeholders. According to the answers to questions two and five, the participants would adopt misuse case diagrams and misuse case specifications in their daily practice. The answers to questions three and six indicate that the notation of misuse case diagrams and the use case specification template enable engineers to properly capture security requirements. The answers to questions 11, 12 and 13 further conclude that RMCM is valuable to capture security requirements.

RQ2

In our questionnaire, we did not include questions explicitly referring to control flow structures because they are perceived by engineers as a feature of the approach, which has been evaluated as being expressive enough to conveniently capture security requirements (question six in Table 4.11).

RQ3

The answers given to questions seven and eight let us respond to *RQ3*. The answers given to these two questions are inconsistent, with two '*agree*', one '*disagree*', and one '*strongly disagree*' for question seven (average score 2.25), and two '*probably*' and two '*probably not*' for question eight (average score 2.5). Therefore, we cannot draw clear conclusions from the data. However, we observed that the responses given by the participants are linked to their software development expertise, with the less experienced software engineers providing the more negative answers.

In general, participants find mitigation schemes less useful than misuse case specifications (the scores of the first six questions are higher). This may be due to less experienced engineers being more reluctant to document their development choices (i.e., the mitigation schemes adopted). In our context, the absence of such documentation makes it difficult to demonstrate compliance with the security standards and regulations.

RQ4

The answers given to question 14 indicate that RMCM-V provides useful assistance for minimizing inconsistencies in the RMCM artifacts of EDLAH2.

The questionnaire study had open, written comments under each section, in which the participants could state their opinions in a few sentences about how RMCM addresses the challenges reported in Section 2.2.2.

Based on the initial comments, we further discussed three aspects with the participants: industrial adoption of the approach, additional extensions in RMCM, and degree of automation.

4.5.2.1 Industrial Adoption of the Approach

Given the current practice in EDLAH2, like in many other environments, there is no systematic way to capture security requirements in use case models. Even though the effort required to apply our modeling approach was considered to be reasonable by EDLAH2 engineers (questions nine and ten in Table 4.11), they stated that it may be a challenge to convince engineers to engage in this additional modeling effort. The costs and benefits of such an activity should be further evaluated to help with adoption. This is, however, a common and general challenge when introducing new practices in software development. For example, in the case of EDLAH2, the proposed methodology enabled the identification of effective test cases capable of identifying 14 vulnerabilities in the developed system. The test cases were derived according to traditional coverage approaches [Jacobson, 2004] that aims to generate a test case for each scenario described in the security use case and misuse case specifications.

4.5.2.2 Additional Extensions in RMCM

The security extensions in RMCM cover various security concerns to be captured in use case models. However, EDLAH2 engineers stated that, due to rapidly changing software and hardware technology, new types of security threats will likely need to be covered with further security extensions. In a way, such extensions can be treated as a knowledge repository of potential vulnerabilities and their associated mitigation schemes. Such repository has to be regularly updated and is expected to help creating awareness of security threats and solutions across an organization.

4.5.2.3 Degree of Automation

RMCM consists of various automated security requirements modeling and specification activities in the context of use case-driven development. Though modeling security requirements in misuse case models is mostly manual, RMCM-V provides automatic consistency checking for these models and feedback to the analyst to help them refine and correct the models. EDLAH2 engineers considered the automated consistency checking of RMCM artifacts to be highly valuable.

4.5.3 Threats to Validity

The main threat to the validity of our evaluation is the generalizability of the conclusions. To mitigate the threat, we applied RMCM to a representative system that includes nontrivial use cases in an application domain entailing numerous and varied security threats. Although we had a relatively low number of respondents in our interviews, we selected the respondents to hold various roles and with substantial industry experience. To limit threats to the internal validity of the case study, we had many meetings with the EDLAH2 engineers to verify the correctness and completeness of our models.

4.6 Conclusion

In this chapter, we presented a use case-driven security requirements modeling method, called RMCM, for documenting the security requirements of Web-oriented software systems in a structured and analyzable form. Our main motivation is to enable security requirements modeling by relying on commonly used artifacts in use-case driven development and by adding a limited number of extensions, thus achieving widespread applicability.

RMCM builds on and integrates existing work and is supported by a tool employing NLP for checking the consistency of artifacts and compliance to the RMCM templates. The key characteristic of our method is that it captures threat scenarios and mitigation schemes in an explicit and structured form, thus enabling both automated analysis of threat scenarios, e.g., consistency and conformance checking, and reuse of mitigation schemes.

Initial results from structured interviews with experienced engineers suggest that RMCM is precise and practical to capture the security requirements of Web-oriented software systems in industrial settings.

In addition to supporting more precise and complete security requirements, RMCM enables the automated generation of test cases for requirements-driven security testing (see Chapter 5). Our ultimate objective is to achieve adequate coverage of the specified security requirements, with traceability information between security requirements and generated test cases.

Chapter 5

Misuse Case Programming

To facilitate communication among stakeholders, software security requirements are typically written in natural language and capture both positive requirements (i.e., what the system is supposed to do to ensure security) and negative requirements (i.e., undesirable behavior undermining security).

In this chapter, we tackle the problem of automatically generating executable security test cases from security requirements in natural language (NL). More precisely, since existing approaches for the generation of test cases from NL requirements verify only positive requirements, we focus on the problem of generating test cases from negative requirements.

We propose, apply and assess Misuse Case Programming (MCP), an approach that automatically generates security test cases from misuse case specifications (i.e., use case specifications capturing the behavior of malicious users). MCP relies on natural language processing techniques to extract the concepts (e.g., inputs and activities) appearing in requirements specifications and generates executable test cases by matching the extracted concepts to the members of a provided test driver API. The evaluation performed with the EDLAH2 case study system provides evidence of the feasibility and benefits of the approach.

5.1 Introduction

Security testing is driven by requirements [Meucci and Muller, 2014] and can be divided in two categories [Tian-yang et al., 2010, Felderer et al., 2016a]: (1) security functional testing validating whether the specified security properties are implemented correctly, and (2) security vulnerability testing addressing the identification of system vulnerabilities. Although several security testing approaches have been proposed [Felderer et al., 2016a], the automated generation of security test cases from security requirements remains limited in industrial settings. Security test cases are manually crafted by engineers who rely on automated tools for a limited set of activities (e.g., input generation to discover SQL injection vulnerabilities [SQL, 2017]).

Most security testing approaches focus on a particular vulnerability (e.g., buffer overflows [Haller et al., 2013, Ognawala et al., 2016] and code injection vulnerabilities [Tripp et al., 2013, Appelt et al., 2014]). These approaches deal with the generation of simple inputs (e.g., strings, files), and cannot be adopted to verify that the system is not prone to complex attack scenarios involving several interactions among parties, e.g., stealing an invitation email to register multiple fake users on a platform. Model-based approaches are capable of generating test cases based on interaction protocol specifications [Veanes et al., 2005, Silva et al., 2008] and thus can potentially generate test cases for such complex attack scenarios [Lebeau et al., 2013]. They require formal models, which limits their adoption in industrial settings. Unfortunately, engineers tend to avoid such models because of the costs related to their development and maintenance, especially in contexts where system requirements in Natural Language (NL) are already available. There are approaches that generate functional system test cases from NL requirements [de Figueiredo et al., 2006, Carvalho et al., 2014, Wang et al., 2015a, Wang et al., 2015b, Kaplan et al., 2008]. However, these approaches can be adopted in the context of security functional testing, but not security vulnerability testing since they generate test cases only for the intended behavior of the system. In contrast, security vulnerability testing deals with the problem of simulating the behavior of a malicious user. Our goal in this chapter is to enable automated security vulnerability test case generation from NL requirements. Our motivation is to have a systematic way to identify threats, to test whether they can be exploited, and to automate testing relying exclusively on artifacts that can be realistically expected in most environments.

In this chapter, we propose, apply and assess Misuse Case Programming (MCP), an approach that generates security vulnerability test cases from misuse case specifications. To generate executable test cases from misuse case specifications we employ some concepts of *natural language programming*, a term which refers to approaches automatically generating software programs (e.g., executable test cases) from NL specifications [Ballard and Biermann, 1979, Pulido-Prieto and Juárez-Martínez, 2017]. To enable the automated generation of executable test cases, MCP assumes that security requirements are elicited according to a misuse case template that includes keywords to support the extraction of control flow information. Our MCP tool currently relies on the Restricted Misuse Case Modeling (RMCM) template (see Section 4.3.2), which presents these characteristics. To interact with the system under test, MCP requires a test driver API that implements basic security testing activities (e.g., requesting a URL).

The natural language programming solution implemented by MCP includes an initial Natural Language Processing (NLP) step in which MCP derives models that capture the control flow of the activities described in a misuse case specification. MCP then translates the derived models into sequences of executable instructions (e.g., invocations of the test driver API's functions) that implement the malicious activities. To this end, we adapt the idea, developed by other works [Manning et al., 2014, Le et al., 2013, Guzzoni et al., 2007, Landhausser et al., 2017], of combining string similarity and ontologies [Gruber, 1993] to generate test cases from misuse case specifications. Similarly to other approaches, MCP builds an ontology that captures the structure of the given test driver API and generates executable instructions by looking for nodes in the ontology that are similar to phrases in NL requirements. The specificity of MCP is that it integrates additional analyses required to enable automated testing, which include the identification of test inputs, the generation of test input values and the generation of test oracles.

We successfully applied and evaluated our approach to an industrial case study in the healthcare domain, thus showing evidence that the approach is practical and beneficial to automatically generate test cases detecting vulnerabilities in industrial settings.

This chapter is structured as follows. The Section 5.2 introduces an overview of the approach. Sections 5.3 to 5.6 provide the details of the cored technical parts of our approach. Section 5.7 describes the MCP toolset. Section 5.8 presents our industrial case study. We conclude the chapter in Section 5.9.

5.2 Overview of MCP

The process in Fig. 5.1 presents an overview of our approach. MCP takes as input a set of misuse case specifications and a test driver API implementing the functions required to test the system (e.g., functions that load URLs). The MCP tool includes a generic test driver API for Web testing that can be extended for system specific operations. The input misuse case specifications should conform to a template which enforces (i) the use of simple sentences to facilitate NLP, (ii) the use of keywords to capture the control flow, and (iii) the use of attack keywords to specify which inputs should be generated according to predefined attack patterns. These are some of the characteristics of the RMCM template, though MCP may work with other templates.

MCP generates as output a set of executable security test cases that rely on the provided test driver API to perform the activities described in the misuse case specifications.

An essential component of the approach is an automatically populated ontology (hereafter *MCP ontology*). An ontology is a graph that captures the types, properties and relationships of a set of individuals (i.e., the basic blocks of an ontology). MCP uses the ontology to model programming language and test infrastructure concepts (Label A in Fig. 5.1), to capture the relationships and structure of the classes of the test driver API (Label B), to capture the relationships between inputs (Label D), and to represent the variables declared in the generated test case (Label F). We employ an OWL ontology [OWL, 2017] instead of UML diagrams because OWL provides simple means to query the modeled data. The MCP ontology is populated and managed using Apache Jena [Jen, 2017].

To generate test cases from misuse case specifications, MCP works in four phases. In the first phase, *Map the test driver API to the MCP ontology*, MCP processes the test driver API and augments the MCP ontology with individuals that model the classes and functions belonging to the given test driver API (Labels A and B). In the second phase, *Generate misuse case models*, MCP relies on an NLP pipeline to derive models that explicitly capture the control flow implicitly described in misuse cases (Label C).

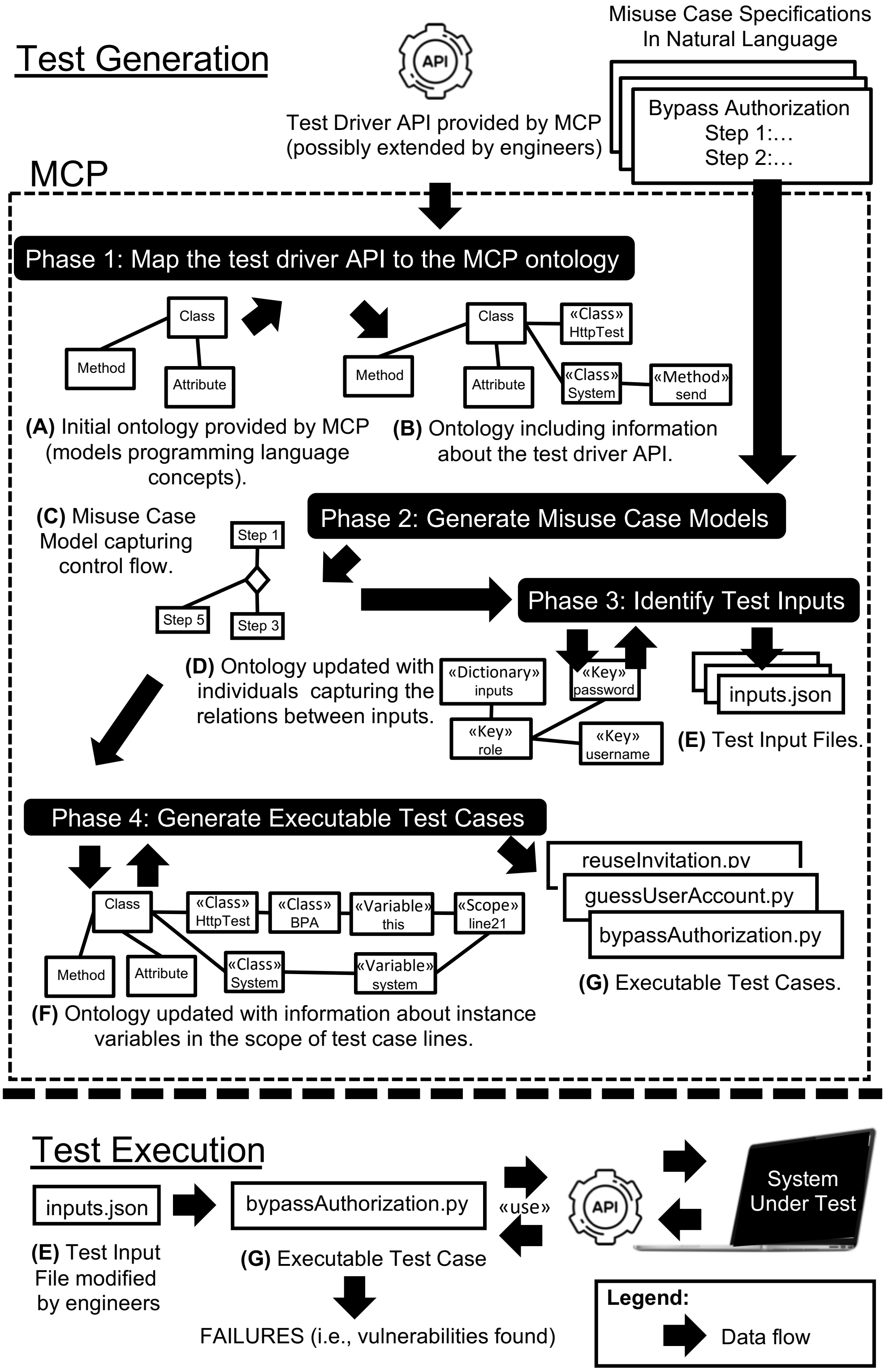

Test Generation
Misuse Case Specifications In Natural Language
Test Driver API provided by MCP (possibly extended by engineers)
Bypass Authorization
Step 1:…
Step 2:…
MCP
Phase 1: Map the test driver API to the MCP ontology
Class
Method
Attribute
«Class» HttpTest
«Class» System
«Method» send
(A) Initial ontology provided by MCP (models programming language concepts).
(B) Ontology including information about the test driver API.
(C) Misuse Case Model capturing control flow.
Step 1
Step 5
Step 3
Phase 2: Generate Misuse Case Models
Phase 3: Identify Test Inputs
(D) Ontology updated with individuals capturing the relations between inputs.
«Dictionary» inputs
«Key» password
«Key» role
«Key» username
inputs.json
(E) Test Input Files.
Phase 4: Generate Executable Test Cases
«Class» BPA
«Variable» this
«Scope» line21
«Variable» system
reuseInvitation.py
guessUserAccount.py
bypassAuthorization.py
(F) Ontology updated with information about instance variables in the scope of test case lines.
(G) Executable Test Cases.
Test Execution
inputs.json
bypassAuthorization.py
«use»
API
System Under Test
(E) Test Input File modified by engineers
(G) Executable Test Case
FAILURES (i.e., vulnerabilities found)
Legend:
Data flow

Figure 5.1. Overview of the MCP approach.

In the third phase, *Identify test inputs*, MCP determines the inputs to be sent to the system. It first identifies the input entities (Label D) and then prepares a configuration file that will be filled out by engineers with concrete input values to be used during testing (Label E). MCP can automatically generate the input values when these values can be derived by relying on predefined strategies (e.g., using grammars to generate inputs for code injection attacks [Tripp et al., 2013, Appelt et al., 2014]).

In the fourth phase, *Generate executable test cases*, MCP automatically generates executable test cases from the misuse case models (Labels C and G). Each generated test case follows the control flow in the corresponding misuse case model and, for each step in the model, executes an operation implemented by the given test driver API. MCP employs a natural language programming solution to map each step in the misuse case model to an operation exposed by the test driver API. This solution maps NL commands (i.e., sentences in misuse case steps) to objects and methods of the provided API by retrieving information from the MCP ontology (Label F). While generating the test cases, the MCP ontology is augmented with individuals matching the variables declared in the test case.

We provide the MCP tool with a set of predefined misuse case specifications derived from the OWASP testing guidelines [Meucci and Muller, 2014]. These misuse cases can be reused (or adapted) across multiple projects; in addition, security analysts can write new, system specific misuse case specifications.

The rest of the chapter provides a detailed description of each phase of MCP shown in Fig. 5.1, with a focus on how we achieved automation. The misuse case *Bypass Authorization Schema* in Fig. 5.2 will be used as an example to demonstrate these phases of MCP. This misuse case specification is produced in the RMCM template (see Section 4.3.2).

5.3 Mapping the Test Driver API to an Ontology

We provide an ontology (i.e., the MCP ontology) with concepts common to object-oriented programming languages (e.g., `Type`, `Class`, `Attribute`, `Method`, and `Parameter`). The MCP ontology also captures the concepts required to model the runtime behavior of a test case: `Instance` (i.e., an instance of a `Type`) and `Variable` (i.e., a program variable pointing to an instance). Fig. 5.3 shows part of the MCP ontology; we model programming language concepts as *types*, shown in Fig. 5.3 using UML classes. We depict the ontology using a UML class diagram. UML classes model types of individuals. UML associations capture properties.

MCP automatically populates the MCP ontology with types and individuals that match the elements (e.g., methods) in the test driver API. For example, Fig. 5.4-A shows part of the populated MCP ontology that models the test driver API used in our case study. UML classes model types. UML objects model individuals. Part-A models the test driver API. Part-B models the variables in the scope of Line 21 of the test case in Fig. 5.9. `System` is a type while `send` is an individual of type `Method` with the property `methodOf` set to `System` (we use association links to model properties).

1 **MISUSE CASE** Bypass Authorization Schema

2 **Description** The MALICIOUS user accesses resources that are dedicated to a user with a different role.

3 **Precondition** For each role available on the system, the MALICIOUS user has a list of credential of users with that role, plus a list functions/resources that cannot be accessed with that role.

4 **Basic Threat Flow**

5 1. FOREACH role

6 2. The MALICIOUS user sends username and password to the system through the login page

7 3. FOREACH resource

8 4. The MALICIOUS user requests the resource from the system.

9 5. The system sends a response page to the MALICIOUS user.

10 6. The MALICIOUS user EXPLOITS the system using the response page and the role.

11 7. ENDFOR

12 8. ENDFOR

13 **Postcondition:** The MALICIOUS user has executed a function dedicated to another user with different role.

14 **Specific Alternative Threat Flow (SATF1)**

15 RFS 4.

16 1. IF the resource contains a role parameter in the URL THEN

17 2. The MALICIOUS user modifies the role values in the URL.

18 3. RESUME STEP 4.

19 4. ENDIF.

20 **Postcondition:** The MALICIOUS user has modified the URL.

21 **Specific Alternative Threat Flow (SATF2)**

22 RFS 4.

23 1. IF the resource contains a role parameter in HTTP post data THEN

24 2. The MALICIOUS user modifies the role values in the HTTP post data.

25 3. RESUME STEP 4.

26 4. ENDIF.

27 **Postcondition:** The MALICIOUS user has modified the HTTP post data.

28 **Specific Alternative Flow (SAF1)**

29 RFS 6

30 1. IF the response page contains an error message THEN

31 2. RESUME STEP 7.

32 3. ENDIF.

33 **Postcondition** The malicious user cannot access the resource dedicated to users with a different role.

Figure 5.2. 'Bypass Authorization Schema' misuse case specification.

Although our approach is language independent, the MCP tool works with the test driver API in Python. Therefore, our tool includes a Python component that relies on reflection to extract the names of classes, methods and method parameters from the given API. Although the Python programming language does not enforce the declaration of method parameter types, we assume that parameter types are captured by means of function annotations, a common practice adopted by Python programmers to document their software [Pyt, 2017].

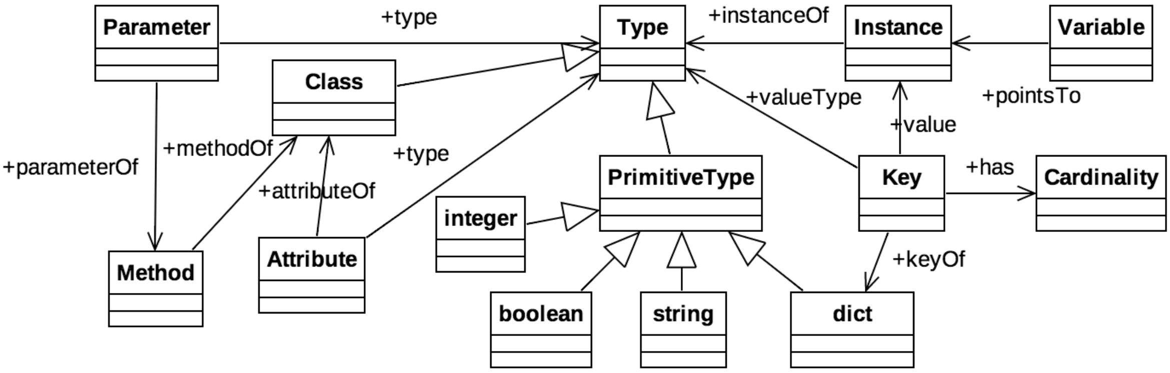


Figure 5.3. Part of the MCP ontology capturing programming language concepts.

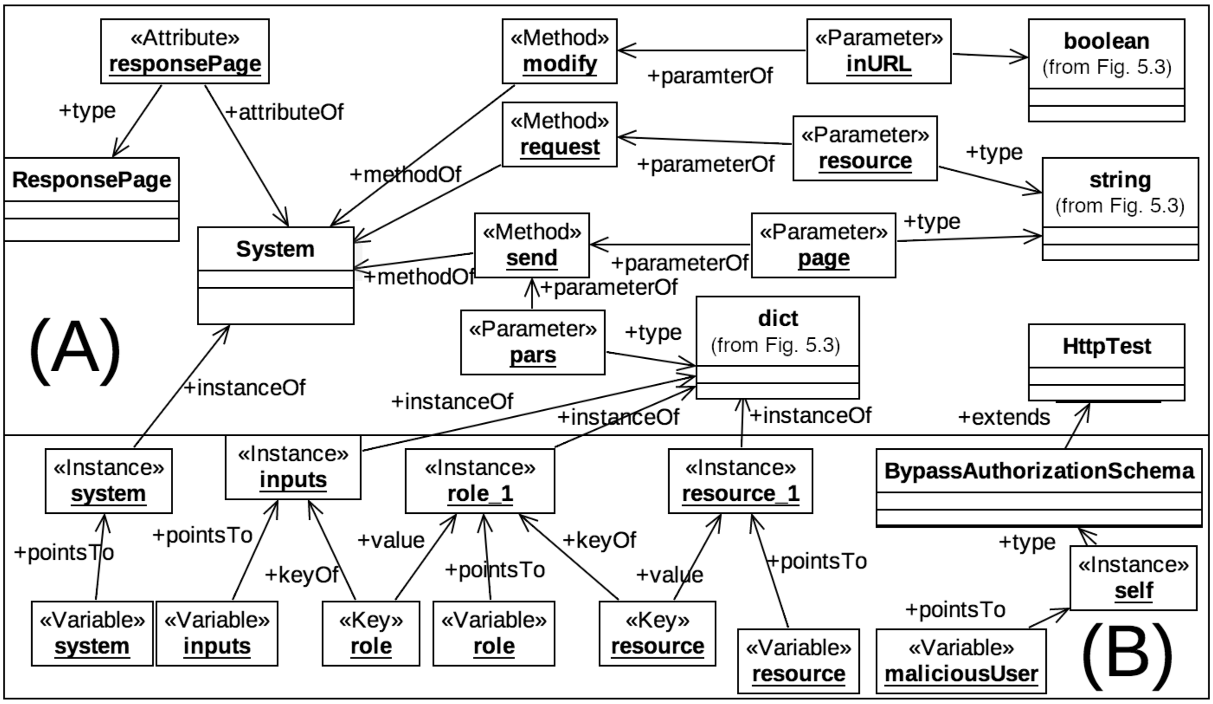


Figure 5.4. Part of the MCP ontology populated when generating a test case for ‘Bypass Authorization Schema’.

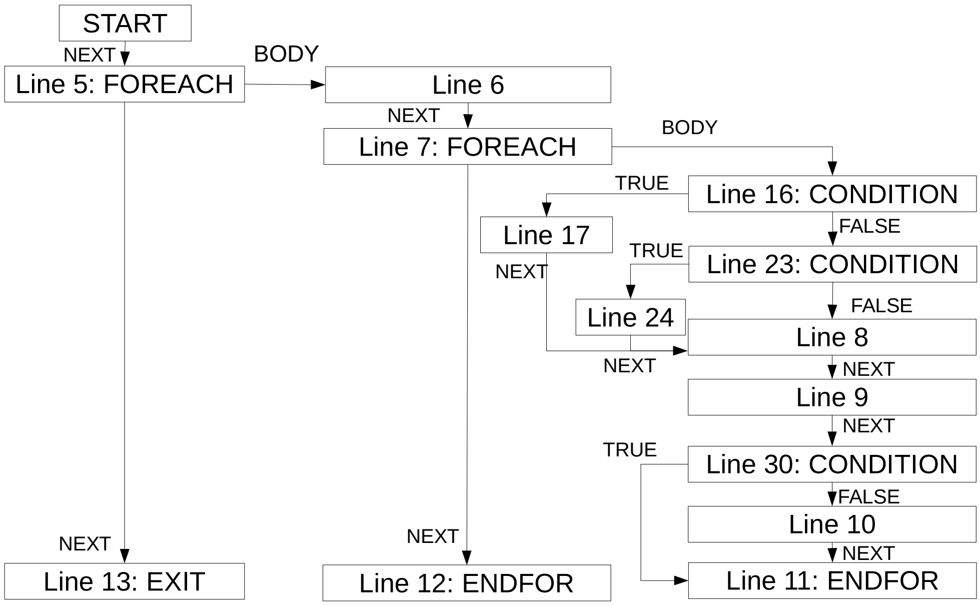


Figure 5.5. Misuse case model for the misuse case specification in Fig. 5.2.

5.4 Generating Misuse Case Models

MCP automatically generates a misuse case model for each misuse case specification (see Fig. 5.5). It employs an NLP pipeline that looks for control flow keywords (e.g., `IF ... THEN`, `FOREACH`, `RFS` and `RESUME` in RMCM) to build a model that captures the control flow implicitly specified in the misuse case specification. Each node in the model corresponds to a step in the misuse case specification. For simplicity, in Fig. 5.5, we indicate only the type of the control flow nodes (i.e., `CONDITION`, `FOREACH`, `ENDFOR` and `EXIT`), while we report the line number of the step for the remaining nodes.

We do not provide details of the misuse case model generation because it is algorithmically simple and similar to the one adopted in the previous work [Wang et al., 2015a]. Briefly, for each condition keyword encountered (i.e., `IF ... THEN`), MCP generates a control flow node that is linked to the first steps of the false and true branches. For each iteration keyword (i.e., `FOREACH` and `DO ... UNTIL`), it generates a node that is linked to the node corresponding to the step in the iteration body (see the arrows `BODY` in Fig. 5.5) and to the node matching the step executed after the iteration (see the arrows `NEXT`).

5.5 Identifying Test Inputs

MCP determines input entities (e.g., ‘role’, ‘password’, ‘username’, and ‘resource’ in Fig. 5.2), input relationships (e.g., each ‘username’ is associated to a ‘role’), and values to be assigned to input entities.

Consistent with RMCM, MCP assumes that input entities appear in misuse case steps with a verb that indicates that one or more entities are sent to the system under test by an actor (e.g., “The malicious user sends username and password to the system” and “The malicious user inserts the password into the system”). SRL (see Section 2.3) is employed to automatically determine sentences indicating the sending of an entity to a destination. Depending on the verb, SRL usually tags destinations with `A2` or `AM-LOC` (see Section 2.3). Therefore, a sentence containing terms tagged with `A2` or `AM-LOC` likely describes an input activity. In such sentences, MCP looks for the entities being sent, which match the terms tagged with `A1` (i.e., the terms affected by the verb).

MCP automatically identifies relationships between input entities to avoid generating input values that may hamper the detection of vulnerabilities. For example, in Fig.5.2, we need to determine the roles associated to a username and password. This is necessary, for instance, to avoid using the username ‘Mr. Phu’, who is a patient, to simulate the behavior of a malicious doctor trying to access managers’ data. If we use the username with the wrong role, the test case may not reveal that the system is vulnerable (e.g., malicious doctors might be able to access managers’ data while patients might not).

MCP relies on the fact that a relationship between input entities can be derived from the control flow in a misuse case specification. For example, in Fig. 5.2, there is a one-to-many relationship between ‘role’ and ‘resource’ because multiple resources are requested with the same role (see Lines 5, 7 and 8). There is a one-to-one relationship between ‘role’ and ‘username’ because only one username is sent for each role (see Lines 5 - 6).

The MCP ontology is employed to capture input relationships by creating instances of the `dict` type. The `dict` type in the ontology is used to model the Python dictionary type, which maps keywords to values. Fig. 5.6 shows part of the populated MCP ontology that captures the input relationships in the misuse case in Fig. 5.2. The `dict inputs` individual contains one `Key` individual for each input entity in the misuse case specifications (e.g., `role`, `username` and `password`). Also, it contains additional `dict` individuals for each entity appearing in an iteration (e.g., `role`) because these entities usually present containment relationships (e.g., each `role` has an associated `username` and `password`).

MCP relies on engineers to select input values. Automating the generation of input values is a challenge since it entails a complete understanding of system specifications. For example, to generate input values from the misuse case in Fig. 5.2, MCP needs existing users and roles, which cannot be automatically extracted without the architecture of the system under test. This information can be manually retrieved by engineers who know the system architecture and configuration.

To guide engineers in generating input values, MCP automatically generates a JSON file using the MCP ontology. The JSON format represents the content of dictionaries in textual form. The generated file contains input types (e.g., `role`) and placeholders to be replaced by engineers with values. Fig. 5.7 shows the JSON file generated from the MCP ontology in Fig. 5.6. Fig. 5.8 shows part of the same file with values. Note that engineers can specify more values for an input entity

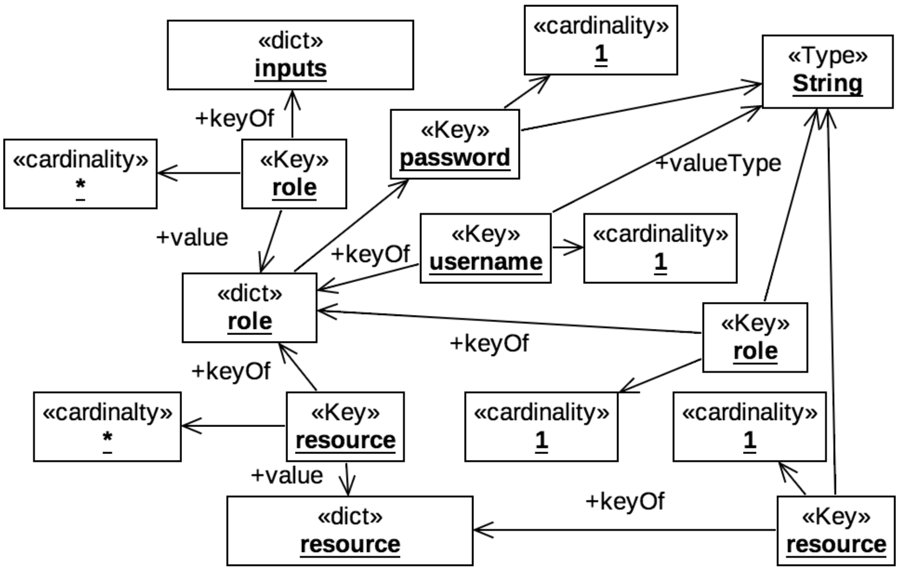


Figure 5.6. Part of the populated MCP ontology for Fig. 5.2.

```
{"role": [
  {
    "password": "REPLACE-THIS-STRING",
    "role": "REPLACE-THIS-STRING",
    "username": "REPLACE-THIS-STRING"
    "resource": [
    {
      "resource": "REPLACE-THIS-STRING",
      "error_message": "REPLACE-THIS-STRING",
      "role_values": "REPLACE-THIS-STRING",
      "the_resource_contains_the_role_parameter_in_the_URL": "PUT-EXPRESSION",
      "the_resource_contains_the_role_parameter_in_the_HTTP_post_data": "PUT...
    },
     ADD-MORE-ENTRIES
    ],
  },
  ADD-MORE-ENTRIES
  ]
}
```

Figure 5.7. Input file generated by MCP.

as suggested by the keyword `ADD-MORE-ENTRIES`; this is necessary to deal with iterations. The repeated entries might have a complex structure like in the case of `role` and `resource` which contain inner values (see Fig. 5.8).

To reveal some vulnerabilities, it is necessary to generate a large number of input values by using a predefined input generation strategy; this happens, for example, in the case of misuse cases that model attacks based on dictionary values or code injection (e.g., SQL injection). To assist engineers in such cases, MCP requires that an input generation strategy be indicated with a keyword in the misuse case specification (e.g., the keyword ‘DICTIONARY VALUES’ in Fig. 2.5).

To determine whether a predefined strategy needs to be used, MCP checks the terms tagged with `A1` (i.e., the entity sent to the system) matching the keyword for the given strategy (e.g., ‘DICTIO-

```
{"role": [
    {
      "role": "Doctor",
      "username": "phu@mymail.lu"
      "password": "testPassword1",
      "resource": [
       {
         "resource": "http://www.icare247.eu/?q=micare_invite&accountID=11"
         "error_message": "error",
    . . .
       },
      {
        "resource": "http://www.icare247.eu/?q=micare_skype/config&clientID=36"
        "error_message": "error",
        "the_resource_contains_the_role_parameter_in_the_URL": False,
        "the_resource_contains_the_role_parameter_in_the_HTTP_post_data": False
      }, ], }
    {
      "role": "Patient",
    . . .
```

Figure 5.8. Part of the JSON file in Fig. 5.7 with input values.

NARY VALUES' in Fig. 2.5). If so, MCP looks for the terms tagged with AM−MNR (see Section 2.3)), which are the input entities to which dictionary values are assigned (e.g., 'username' and 'password' in the example above).

5.6 Generating Executable Test Cases

MCP generates an executable test case for each misuse case specification. In the MCP tool, each generated test case corresponds to a Python class that implements a method named `run`. Fig. 5.9 shows part of the test case generated for the misuse case in Fig. 5.2.

MCP declares and initializes three variables, `system`, `maliciousUser` and `inputs` (Lines 3, 4 and 5 in Fig. 5.9). The variable `system` refers to an instance of the class `System`, which provides methods that trigger the functions of the system under test (e.g., `request`). The variable `maliciousUser` refers to the test class, since the test class simulates the behavior of the malicious user. The variable `inputs` refers to a dictionary populated with the input values specified in the JSON input file. These three assignments are given in the MCP ontology with the individuals `maliciousUser`, `system` and `inputs` (see Fig. 5.4-B).

MCP identifies the program elements (e.g., an API method) to be used in the generated test case based on string similarity. To do so, we employ a string similarity solution successfully used in the prior work [Wang et al., 2018], i.e., a function based on the Needleman-Wunsch string alignment algorithm [Needleman and Wunsch, 1970]. To generate an executable test case, MCP processes all the nodes in the generated misuse case model (see Fig. 5.5). For each control flow node, MCP generates a control operation in the test case. For each other node, MCP generates both a method call and an assignment instruction by using the string similarity, and then selects one of them according to a scoring procedure.

```
class bypassAuthorizationSchema(HTTPTester):
 def run(self):
  system = System(path=self.rootPath)
  maliciousUser = self
  inputs = self.loadInput("inputs.json")
  roleIter = inputs["role"].__iter__()
  while True:
   try:
    role = roleIter.__next__()
    parameters = dict()
    parameters["password"] = role["password"]
    parameters["username"] = role["username"]
    system.send("login page",parameters)
    resourceIter = role["resource"].__iter__()
    while True:
     try:
      resource = resourceIter.__next__()
      if not eval(resource["the_resource_contains_a_role_
        parameter_in_the_URL"]):
       if not eval(resource["the_resource_contains_a_role_parameter..
        system.request(resource)
        maliciousUser.responsePage = system.responsePage
        if not responsePage.contains( resource["error message"] )
         parameters = dict()
         parameters["resource"] = resource["resource"]
         parameters["role"] = role["role"]
         system.exploit(parameters)
        else:
         maliciousUser.abort("The MALICIOUS user CANNOT ex...")
```

Figure 5.9. Part of the test case generated from the misuse case in Fig. 5.2.

In the following, we present the string similarity solution adopted by MCP, and the generation of method calls, assignments, control flow instructions and oracles.

5.6.1 String Similarity Measures

The Needleman-Wunsch string alignment algorithm maximizes the matching between characters by allowing for some degree of misalignment between them. The similarity degree adopted by MCP is computed as the percentage of matching characters in the aligned strings. In the rest of the chapter, we write that a string s_a belonging to a set of strings S best matches a string s_t if the following holds:

$$\forall s : s \in S, D(s_a, s_t) \geq D(s, s_t) \text{ and } D(s_a, s_t) \geq T$$

with D being the function for computing the degree of similarity of two strings and T being a threshold, set to 0.4 in our experiments, below which matching strings are excluded.

5.6.2 Generation of Method Calls

For each misuse case step, MCP aims to generate a method call that performs the activity described in the sentence. To achieve this goal, MCP must select the correct method to be invoked (i.e., a method with a proper name and parameters that belongs to a specific class instance) and identify which instance variables should be passed as argument.

To identify the class instance that should expose the method to be invoked, MCP queries the MCP ontology looking for individuals that best match, using similarity scores, the actors typically involved in a misuse case sentence: the system, the actor that performs the activity (SRL label `A0`), the actor affected by the activity (SRL label `A1`) or the actor that receives the inputs mentioned in the sentence (SRL labels `A2` and `AM-LOC`). For each selected individual, MCP looks for a method that is most likely to perform the activity described in the misuse case sentence.

MCP selects the method that maximizes a score that results from the average of: (S1) the string similarity degree between the method name and the verb in the sentence (to measure how well the method name matches the activity described by the sentence); (S2) the average string similarity degree of all the parameters with the best matching input entity (to determine if the method is supposed to work with the input entities appearing in the sentence); (S3) the percentage of terms (i.e., verb and noun phrases appearing in the misuse case sentence) that match the method parameters (to measure the completeness of the generated instruction, i.e., to what extent the concepts appearing in the sentence are reflected in the method call). The last point distinguishes MCP from other natural language programming solutions (e.g., [Landhausser et al., 2017]) since these do not measure the completeness of the generated instruction. MCP may also select a method call that best-matches the full sentence; this is done to properly deal with sentences describing specific attacks (e.g., "execute a network sniffing tool" which is implemented by the method `executeNetworkSniffingTool`, whose name includes the verb and the object in the sentence).

After identifying a method as the best candidate for a misuse case sentence, MCP generates the corresponding executable instruction as follows. First, MCP generates the program code corresponding to the invocation of the selected method (e.g., `system.request` in Line 21 in Fig. 5.9). Then MCP identifies the instance variables to pass as arguments; to this end, MCP queries the ontology looking for instance variables with the same type as the method parameter and with the best matching name. For example, in the case of Line 21 in Fig. 5.9, MCP selects the instance variable `resource`, which exactly matches the name of the parameter of the method `request`. If there is no variable matching the method parameter, MCP derives the value to be used directly from the text of the input entity that best matches the parameter name. This is necessary because the misuse case specification may include some of the values to be used during testing. If the parameter is a string, MCP sets the value to the name of the input entity (e.g., `"login page"` in Line 13). If the parameter is a boolean, MCP sets its value to `True` (this helps dealing with methods presenting boolean flags, e.g., the method `modify` shown in Fig. 5.4). Otherwise, MCP signals the missing parameter using a dedicated keyword in the generated code.

MCP also deals with API methods that declare an arbitrary number of parameters. This is the case of method `send` of the class `System` (Fig. 5.4-B), which is used to send a set of input values to the system under test and enables the specification of inputs to be sent according to the input page. For example, a login page may require two inputs (e.g., `username` and `password`), while the page for registering a new user may require more inputs. In Python, an arbitrary number of named arguments can be passed to a method by using a dictionary parameter. For this reason, in the presence of a dictionary parameter whose name does not match any input entity, MCP assumes that the dictionary parameter can be used to pass named arguments to the method. More precisely, MCP uses the identified dictionary parameter to pass input entities that do not match any other method parameter. These entities are taken into account when computing the score of the method (point S3 above). This is what occurs when MCP processes Line 6 of the misuse case specification in Fig. 5.2, which leads to generating the code appearing in Lines 10 - 13 in Fig. 5.9. The parameter `pars` of the method `system.send` is used by MCP to pass additional parameters to the method (i.e., `username` and `password`).

To simplify testing further, in the presence of test driver API methods requiring specific configuration parameters (e.g., the method `System.send` requires a mapping between a page name and its URL), engineers, instead of manually crafting a configuration file, can provide API methods that are invoked by MCP to automatically generate a file with the required configuration parameters.

5.6.3 Generation of Assignments

Assignment instructions are expected to be generated when some data (input or output) is exchanged between an actor and the system under test. MCP relies on SRL to identify the actor who performs the action (i.e., `A0` which is supposed to be the source of the data) and the final destination, which is captured by the SRL labels `A2` or `AM-LOC`. The data being transferred correspond to the terms tagged with `A1`. The assignment is then generated by looking for two instance variables that best match the terms tagged with `A0` (i.e., the data source for the right-hand side) and `A2` or `AM-LOC` (i.e., the destination for the left-hand side). The term tagged with `A1` (i.e., the data being moved) should then match an attribute of the objects referred by the selected variables. For example, the misuse case step “The system sends a response page to the malicious user” (Line 9 in Fig. 5.2) leads to the generation of the assignment in Line 22 in Fig. 5.9.

The score of the generated assignments is calculated by computing the mean of (1) the average string similarity degree for the terms used to identify the left-hand side and right-hand side of an assignment (to measure the likelihood that the selected terms match the concepts in the sentence) and (2) a value indicating the proportion of terms of the misuse case step that appear in the generated assignments (to measure the completeness of the generated assignments with respect to the concepts appearing in the step).

5.6.4 Generation of Control Flow Instructions

The generation of control flow instructions is straightforward and follows typical practices adopted in DSL-based and model-based code generation [Bettini, 2016]. In this section, we focus on the generation of instructions from iterations and conditional sentences in NL, which is not addressed by DSL-based and model-based approaches.

Since iterations (i.e., sentences containing the keyword `FOREACH`) are used to indicate that a sequence of activities is repeated for a given set of input entities, MCP generates a block of code that iterates over all the values of the input variable matching the input entity mentioned in the `FOREACH` sentence. For example, Lines 6 - 9 in Fig. 5.9 show that the test case iterates over the elements of the list named `role`.

Condition sentences, instead, are used to indicate that certain activities are performed when a given condition, written in NL, holds. In general, a condition in the test code can be used to evaluate the truth value of either runtime data (e.g., the value returned by a method call) or input data (e.g., a configuration parameter). To deal with the first case, MCP generates a method call that best matches the condition in NL (Line 23 in Fig. 5.9). If the condition sentence does not match any method call, MCP assumes that the condition works with test input parameters, and thus generates a condition instruction evaluating the truth value of an input entity that matches the sentence (Line 18 in Fig. 5.9). The name of the input entity is added to the JSON input file (see the parameters starting with `the_resource_contains` in Fig. 5.7).

5.6.5 Generation of Oracles

In executable test cases, an automated oracle is typically implemented by means of instructions that report a failure when a certain condition does not hold; this is, for example, what JUnit assertions do [JUn, 2017]. *MCP automatically generates oracles*; this is implicitly achieved during the generation of the executable test case because MCP generates code that matches all the use case steps, including conditions that check erroneous outputs (e.g., Line 30 in Fig. 5.2) and instructions indicating that the malicious user can exploit a vulnerability (e.g., Line 10 in Fig. 5.2). However, to determine if the system in is a legal state or if it generates a valid output, MCP often requires the specification of regular expressions matching conditional sentences in the misuse case specifications (e.g., Line 18 in Fig. 5.9), which, based on our experience is of limited costs, but are still required to be manually specified by engineers.

For example, the condition instruction in Line 23 of the test case in Fig. 5.9 corresponds to Line 30 in Fig. 5.2 and determines whether the system was not able to detect an unauthorized access. The instruction in Line 27 of Fig. 5.9, which corresponds to Line 10 in Fig. 5.2, is used to report a failure. In the MCP tool, the method `System.exploit`, which matches misuse case steps indicating that a malicious user exploits a vulnerability, is used to report a failure.

5.7 Tool support

We have implemented MCP as a Java application. The MCP tool takes as inputs the test driver API and misuse case specifications and automatically generates executable test cases in Python, input files, and configuration files. Fig. 5.10 shows the architecture of MCP tool.

SpecificationProcessor parses misuse case specifications to extract a model (misuse case model) that captures the control flow in the specifications. *OntologyLoader* maps the test driver API into an OWL ontology to have a structured representation of the API elements (i.e., classes, methods, and parameters).

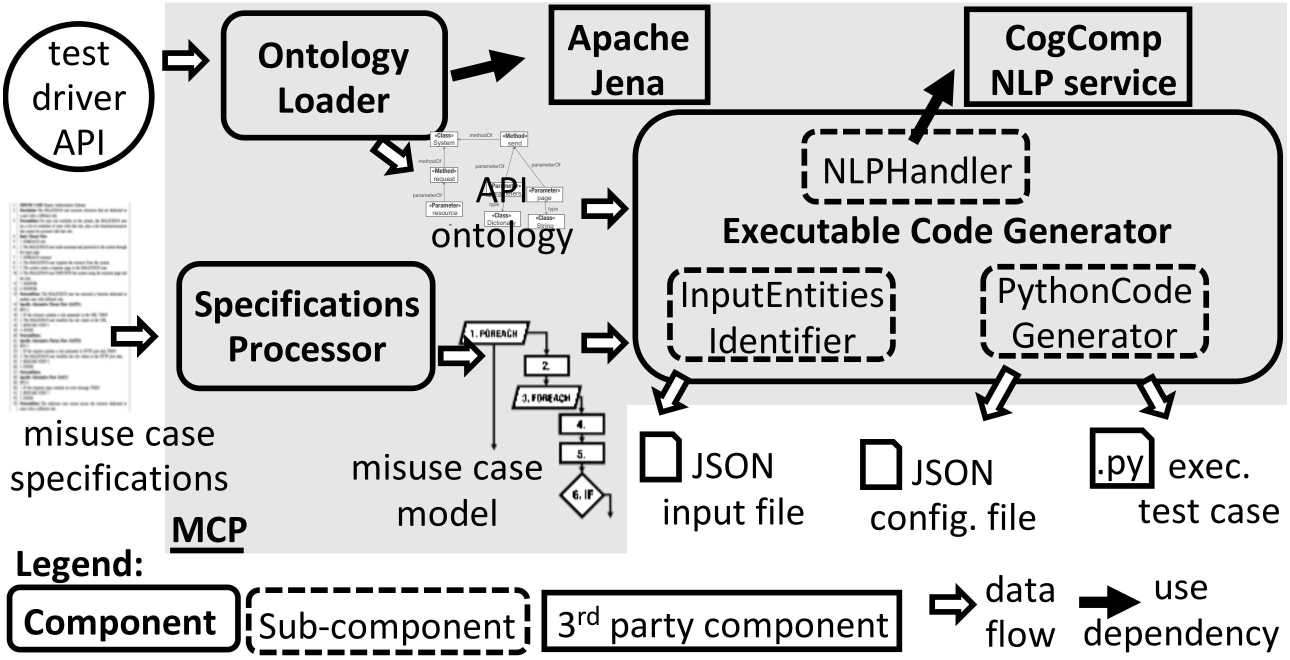


Figure 5.10. MCP Architecture

ExecutableCodeGenerator identifies input entities and generates executable code. *NLPHandler* executes NLP on each misuse case step. More specifically, *NLPHandler* executes the *CogComp NLP pipeline* [University of Illinois, 2017] to perform Semantic Role Labeling. *InputEntitiesIdentifier* uses SRL results to identify input entities. To speed up NLP, *NLPHandler* relies on the CogComp NLP pipeline running as a Web service[1]. If the service is not reachable, the analysis is executed locally.

PythonCodeGenerator generates Python code together with configuration files. It processes the misuse case model and generates a method call or an assignment for each use case step, except condition and iteration steps which are translated into Python instructions. In general, thanks to SRL outputs, a method call is identified by selecting a method (1) that belongs to a class instance with a name similar to either the actor performing the activity or the destination in the sentence, (2) that has a name textually similar to the verb in the sentence, and (3) that has parameters matching the remaining semantic roles in the sentence. An assignment instruction is generated when some data is exchanged between an actor and the system. It is generated by looking for two variables textually similar to the source and destination of the data exchange.

Additional details about MCP, including executable files and a screencast, are available on the tool's website at: **`https://sntsvv.github.io/MCP/`**.

5.8 Empirical Evaluation

We have performed an empirical evaluation to respond to the following research questions:

[1] MCP can rely both on the installation of the University of Pennsylvania, http://macniece.seas.upenn.edu:4001/annotate, or services running in-house.

- RQ1. Does MCP correctly identify input entities?
- RQ2. Can MCP generate executable test cases from misuse case specifications?
- RQ3. How do the generated test cases compare to manual test cases in terms of effectiveness, soundness and costs?

5.8.1 Case Study System and Empirical Setup

We applied MCP to generate test cases for the software system developed in the context of the EU project EDLAH2 [EDLAH2, 2017b] and the open-source continuous integration server Jenkins [Eclipse Foundation, 2020] (see Section 2.1).

The EDLAH2 engineers follow the RMCM methodology to capture security requirements because RMCM specifications are written in NL and thus ease communication among all the stakeholders. The EDLAH2 misuse case specifications include a total of 68 misuse cases which describe both general attack patterns derived from the OWASP guidelines [Meucci and Muller, 2014, OWA, 2018] and system specific attacks that leverage some characteristics of the EDLAH2 system. For example, one of the EDLAH2 misuse cases models a malicious user who generates multiple user accounts by stealing the token of the page for inviting new users. The misuse case specifications of the EDLAH2 system have been used to manually derive test cases (scripts for manual testing and executable test cases).

We have used the MCP tool to generate executable test cases from 12 misuse case specifications. We have selected 12 misuse cases targeting the Web interface and with the highest risk according to the OWASP risk rating methodology [Meucci and Muller, 2014]. Nine of the test cases manually derived from the selected misuse cases enabled the identification of vulnerabilities.

To ensure the generalizability of the results, we also applied the 12 generated test cases to test the Jenkins case study system. We target 10 vulnerabilities of Jenkins. Seven out of these 10 vulnerabilities were reported in the CVE database [MITRE Corporation, 2020] in the second half of 2018 (i.e., CVE-2018-1000406 [MITRE, 2018a], CVE-2018-1000409 [MITRE, 2018b], CVE-2018-1999003 [MITRE, 2018c], CVE-2018-1999004 [MITRE, 2018d], CVE-2018-1999006 [MITRE, 2018e], CVE-2018-1999046 [MITRE, 2018g], and CVE-2018-1999047 [MITRE, 2018h]). The other three vulnerabilities are related to the default configuration of Jenkins such as the lockout mechanism and the weak password requirement. Besides the 12 generated test cases derived from OWASP testing guidelines [Meucci and Muller, 2014] and EDLAH2 case study, we derived five additional misuse case specifications from five selected Jenkins vulnerabilities reported in CVE database and generated test cases with MCP. To summarize, we executed 17 MCP-generated test cases for the Jenkins case study.

To perform the experiments, we have used a test driver API that was developed to support the manual implementation of the test cases from the EDLAH2 misuse case specifications. The API consists of ten classes and 87 methods in total.

5.8.2 RQ1 - Input Entities Identification

MCP reports the input entities in the `JSON` input file, which we inspected to evaluate the capability of MCP to determine correct input entities. We measure precision and recall according to standard formula [Lane, 2003]. In our context, true positives coincide with input entities, identified by MCP, which are correct (i.e., necessary to perform the test). False positives are input entities that do not correspond to software inputs. False negatives are input entities required to perform the attack (e.g., an input that should be provided to a form field of a Web page), which have not been identified by MCP.

In total, MCP leads to 86 true positives (i.e, input entities correctly identified), one false positive, and 10 false negatives. The false positive is due to the fact that one input entity belongs to an activity that is executed under conditions that do not hold for the EDLAH2 system (this is the case of the input entity 'role values' which is used in 'Bypass Authorization Schema' only for systems with URLs including role parameters). The three false negatives are caused by a concept (i.e., `invitation request`) which corresponds to three distinct input entities for the system under test (i.e., `email`, `username` and `message`). Other two false negatives are related to `password` and `password confirmation` input entities in the EDLAH2's reset password functionality. Five false negatives are caused by the sign up activity in Jenkins which corresponds to five input entities (i.e., `username`, `password`, `password confirmation`, `full name`, and `email`). Overcoming false positives and negatives has shown to be simple since we did not modify the generated test code, but simply removed and added entries from and to the JSON input file. *Precision and recall are particularly high*, 0.99 and 0.90 respectively, which will favor the adoption of the technique in industrial settings.

5.8.3 RQ2 - Test Cases Generation

We inspected the source code of the generated test cases to spot the presence of errors affecting the control instructions, assignments, method calls and parameters. We also counted the number of test cases successfully executed without runtime errors due to programming mistakes. To execute the test cases, we have filled out the MCP input files for EDLAH2 and Jenkins case study systems.

The test cases generated by MCP do not contain any programming error and, furthermore, were all successfully executed against EDLAH2and Jenkins systems. The generated test cases, one for each misuse case, are not trivial, and include a total of 1107 lines of code (101 max for a single test, 54 min), 252 method calls (23 max, 10 min), 71 assignments (7 max, 3 min), 358 method arguments (42 max, 12 min). A subset of 230 method invocations concern the test driver API methods, while the rest corresponds to general Python utility methods. The generated test cases have been delivered to our industrial partners and are used to test the EDLAH2 system.

5.8.4 RQ3 - Comparison Between Automatically Generated and Manual Implemented Test Cases

To answer the RQ3, we compared 12 test cases automatically generated by MCP with the test cases manually derived by EDLAH2 engineers for the same set of misuse case specifications, and with respect to effectiveness, soundness, and costs.

A security test case is *effective* if it is capable of discovering vulnerabilities that affect the system and it is *sound* if it does not report false alarms. The test cases generated by MCP identified all the nine vulnerabilities detected with manual testing, which shows that MCP test cases are as effective as manual test cases. Note that all these vulnerabilities result from real errors committed by engineers during software development. The test cases generated by MCP did not lead to the identification of any false alarm, thus showing that the approach is sound.

We discuss *costs* by comparing the effort required to perform vulnerability testing using MCP with the effort required by manual testing. To manually implement executable test cases, engineers must read and understand the security specifications of the system, an activity that requires substantial effort. Also, the implemented test cases might be error-prone and difficult to maintain. In the case of MCP, engineers do not need to implement or maintain executable test cases, but they require a test driver API and security specifications in NL. The results reported in Chapter 4 have shown that experienced engineers find that writing security specifications according to a structured format is helpful to support communication among stakeholders, which motivates the adoption of the RMCM methodology. In the presence of RMCM specifications, the generation of vulnerability test cases can be fully automated by MCP. To give additional evidence of the benefits of MCP, we count the lines of code of nine test cases developed by EDLAH2 engineers based on nine misuse case specifications, which is 1523. Considering that EDLAH2 requirements include more than 60 misuse cases, the manual implementation of all the required test cases would become expensive because of the effort required to write hundreds of lines of code after carefully reading several requirements specifications. This further motivates the adoption of MCP.

A test driver is also required by the manually written test cases, including functional test cases. Since a project-specific test driver API is necessary for both functional and security testing, its development costs do not directly result from the adoption of MCP. In addition, we provide the MCP tool with a general test driver API that can be used with different Web projects, thus further reducing API development costs.

In both MCP and manual testing, engineers need to identify the input values to be used during testing (e.g., URLs). In general, the number of input values required for MCP and manual test cases derived from the same set of misuse cases is similar since they both cover the same scenarios. For each of the 12 MCP test cases generated in our experiment, engineers provided, on average, 15 distinct input values (excluding dictionary values) in the JSON input files and 11 configuration parameters required by the test driver API methods.

5.8.5 Threats to Validity

The main threat to the validity regards generalizability, since results are linked to case study systems considered and the selected misuse case specifications. To deal with this threat, we selected case study systems that are representative of modern Web systems but are different from both a technical and process perspective. Moreover, we considered misuse cases that enabled the detection of vulnerabilities caused by real mistakes reported in the CVE database [MITRE Corporation, 2020].

5.9 Conclusion

In this chapter, we presented MCP, an approach that automatically generates vulnerability test cases, that is test cases simulating attacks and aimed at uncovering security vulnerabilities. MCP focuses on contexts where security requirements are written in Natural Language (NL), which is a common case since NL facilitates communication among stakeholder as in our industrial case study.

MCP requires as input a set of misuse case specifications and a test driver API and automatically generates a set of executable test cases that simulate the activities described in the misuse case specifications. MCP is a natural language programming solution that automatically translates each step in the misuse case specifications into executable instructions. The identification of the instructions to execute relies on NLP techniques. These techniques enable the identification of concepts that match the elements of the test driver API to be used in the test cases. For example, the actor performing an activity usually corresponds to an instance of an API class that exposes a method matching the verb in the sentence. The matching between concepts in NL requirements and the API is enabled by string similarity and an ontology which is used to model the test driver API and the generated test case. MCP assumes a consistent use of terminology between misuse case specifications and test driver API, which is generally true for modern test-driven development approaches. Future work will include the handling of synonyms (e.g., [Wang et al., 2018]).

Empirical results with the EDLAH2 system, which is a representative commercial case study system in the healthcare domain, include the automated identification of real vulnerabilities in the developed system, an indication of the effectiveness of MCP. Also, MCP reduces the effort required for performing security vulnerability testing since it automates the generation of executable test cases which are not trivial to manually implement. The main limitation of MCP is the need for the manual specification of inputs and the need for regular expressions used to process system outputs and determine if the system output is valid. Such limitations are partially addressed by the metamorphic security testing approach described in Chapter 6 and they are further discussed in Chapter 7.

Chapter 6

Metamorphic Security Testing for Web Systems

Security testing verifies that the data and the resources of software systems are protected from attackers. Unfortunately, it suffers from the oracle problem, which refers to the challenge, given an input for a system, of distinguishing correct from incorrect behavior. In many situations where potential vulnerabilities are tested, a test oracle may not exist, or it might be impractical due to the many inputs for which specific oracles have to be defined.

In this chapter, we propose a metamorphic testing approach that alleviates the oracle problem in security testing. It enables engineers to specify metamorphic relations (MRs) that capture security properties of the system. Such MRs are then used to automate testing and detect vulnerabilities.

We provide a catalog of 22 system-agnostic MRs to automate security testing in Web systems. Our approach targets 39% of the OWASP security testing activities not automated by state-of-the-art techniques. It automatically detected 11 out of 14 vulnerabilities affecting three widely used systems, one commercial and two open source systems (i.e., Jenkins and Joomla).

6.1 Introduction

In contexts where test case execution is automated, an automated *test oracle* (i.e., a mechanism for determining whether a test case has passed or failed) is needed to check the execution result. It often consists of comparing expected and observed outputs.

Security test cases seldom rely on automated test oracles, most often because it is infeasible or impractical to specify them due to a large number of test inputs. In other words, *security testing suffers from the oracle problem* [Barr et al., 2015, Staats et al., 2011, Pezze and Zhang, 2014], which refers to situations where it is extremely difficult or impractical to determine the correct output for a given test input. For instance, a security test case for the bypass authorization schema vulnerability should verify, for every specific user role, whether it is possible to access resources that should be available only to a user who holds a different role [Meucci and Muller, 2014]. This type of vulnerability can often be discovered by verifying the access to various resources with different privileges and roles. This is the case for the MCP test case in Fig. 5.2 of Chapter 5, which requires the specification of the URLs that should not be accessed by a user along with a regular expression to determine if the expected error page is returned. Such inputs might be expensive to be produced. Indeed, questions arise when defining oracles. What are the resources that can only be accessible by a user with a specific role or privilege? Are the test outputs consistent with expectations regarding accessibility? In practice, it is not always feasible to answer such questions when expected outputs need to be identified for a large set of test inputs (e.g., for various resources, roles and privileges). Recent incidents involving corporate Web sites, such as Facebook's, indicate that it is particularly difficult to verify, at testing time, large sets of input sequences including the ones that trigger vulnerabilities [Rosen, 2018, Deahl, 2018]. Another example, the case study EDLAH2 provides many types of roles such as administrator, master carer, normal carer, and client. Applying MCP (see Chapter 5) to run the test case *Bypass Authorization Schema* against this system, when we prepare inputs and oracles for test cases we need to enumerate all resources (i.e., URLs dedicated to access functions of the system) which should not be accessed by each type of role. For instance, for the role *normal carer*, we need to list all resources only dedicated to administrators or master carer (e.g., URLs relevant to invitation of new clients). This process requires a lot of efforts from test engineers and is error-prone.

Although several security testing approaches have been proposed, they typically do not address the oracle problem and assume the availability of an implicit test oracle [Barr et al., 2015]. Furthermore, most approaches focus on a particular vulnerability (e.g., buffer overflows [Haller et al., 2013, Ognawala et al., 2016]) and can only uncover vulnerabilities that prevent a system from providing results (e.g., system crashes because of buffer overflows).

Metamorphic Testing (MT) is a testing technique which has shown, in some contexts, to be very effective to alleviate the oracle problem [Chen et al., 1998, Liu et al., 2014]. *MT is based on the idea that it may be simpler to reason about relations between outputs of multiple test executions, called metamorphic relations (MRs), than it is to specify its input-output behavior* [Segura et al., 2016]. In MT, system properties are captured as MRs that are used to automatically transform an initial set of test inputs into follow-up test inputs. If the outputs of the system under test for the initial and follow-up test inputs violate the MR, it is concluded that the system is faulty (see Section 2.5 for additional details).

Considerable research has been devoted to developing MT approaches for application domains such as computer graphics (e.g., [Mayer and Guderlei, 2006, Guderlei and Mayer, 2007, Just and Schweiggert, 2009, Kuo et al., 2011b]), Web services (e.g., [Chan et al., 2007b, Sun et al., 2011, Zhou et al., 2012]), and embedded systems (e.g., [Tse and Yau, 2004, Chan et al., 2007a, Kuo et al., 2011a, Jiang et al., 2013]). Unfortunately, only a few approaches target security aspects [Chen et al., 2016]; also, their applicability is limited to the functional testing of security components (e.g., code obfuscators [Chen et al., 2016]) or to the verification of specific security bugs (e.g., heartbleed [Synopsys Inc., 2018]). They do not support the specification of general security properties by using MRs.

Although MT is automatable, very few MT approaches provide proper tool support [Segura et al., 2016]. This is also a significant obstacle for tailoring the current approaches for security testing. Our goal in this chapter is to adopt MT to address the test oracle problem in security testing. Our motivation is to have a systematic way to specify MRs that capture security properties of Web systems (i.e., properties that are violated only if the system is vulnerable) and to automate security testing by relying on these MRs. An example of MR to spot bypass authorization schema vulnerabilities is: *a Web system should return different responses to two users when the first user requests a URL that is provided to her by the GUI (e.g., in HTML links) and the second user requests the same URL but this URL is not provided to her by the GUI.* In other words, a user should not be able to directly access URLs not provided by the GUI.

In this chapter, we propose an MT approach (hereafter MST– Metamorphic Security Testing) that supports engineers in specifying MRs to capture security properties of Web systems and that automatically detects vulnerabilities (i.e., violations of security properties) based on those relations. Our approach is built on top of the following novel contributions: (1) a Domain-Specific Language (DSL) for specifying MRs for software security, (2) a catalog of system-agnostic MRs targeting well-known security vulnerabilities of Web systems [Meucci and Muller, 2014], (3) a framework that automatically collects the data required to perform MST, and (4) a testing framework that automatically performs security testing based on the MRs and the collected data. To facilitate the specification of MRs in our DSL, we provide an editor which has been implemented as a plug-in for the Eclipse IDE [Ecl, 2018].

We applied our approach to discover vulnerabilities in a commercial Web system, in Jenkins, a leading open source automation server [Eclipse Foundation, 2020], and in Joomla, a popular open source content management system [Joo, 2020]. The approach automatically detected 100%, 75%, and 100% of the targeted vulnerabilities affecting these three systems, respectively. Furthermore, MST has shown to help discover a new vulnerability (i.e., CVE-2020-2162 [Sto, 2020]) in Jenkins. Based on these results and an assessment of the effort involved, we conclude that our approach is practical and beneficial to alleviate the oracle problem in security testing and to automatically detect vulnerabilities in industrial settings. Our MST toolset and the empirical data are publicly available [Web, 2019].

This chapter is structured as follows. In Section 6.2, we present an overview of the approach. Sections 6.3 to 6.6 describe the core technical solutions. Section 6.7 presents our catalog of MRs. Section 6.8 introduces the MST toolset. In Section 6.9, we present the empirical evaluation of our approach. We conclude this chapter in Section 6.10.

6.2 Overview of the Approach

The process in Fig. 6.1 presents an overview of our approach. In Step 1, the engineer selects, from a catalog of predefined MRs, the relations for the system under test. We have derived our catalog of MRs from the testing guidelines [Meucci and Muller, 2014] edited by OWASP [OWA, 2017b]. In

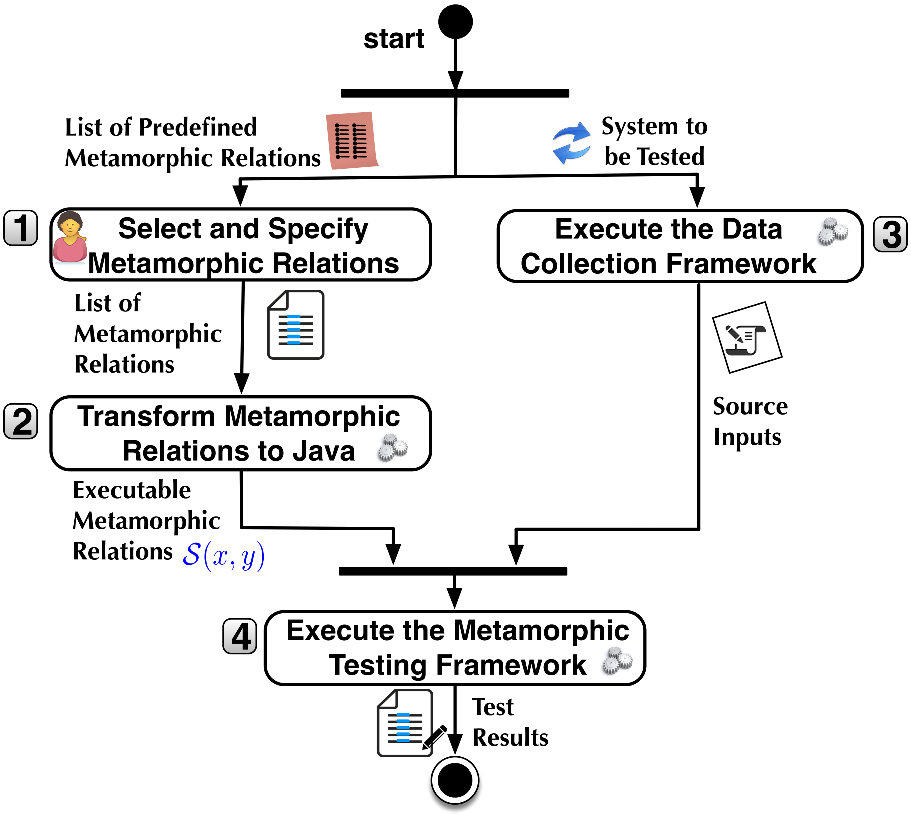

start
List of Predefined Metamorphic Relations
System to be Tested
1
Select and Specify Metamorphic Relations
3
Execute the Data Collection Framework
List of Metamorphic Relations
Source Inputs
2
Transform Metamorphic Relations to Java
Executable Metamorphic Relations $S(x, y)$
4
Execute the Metamorphic Testing Framework
Test Results

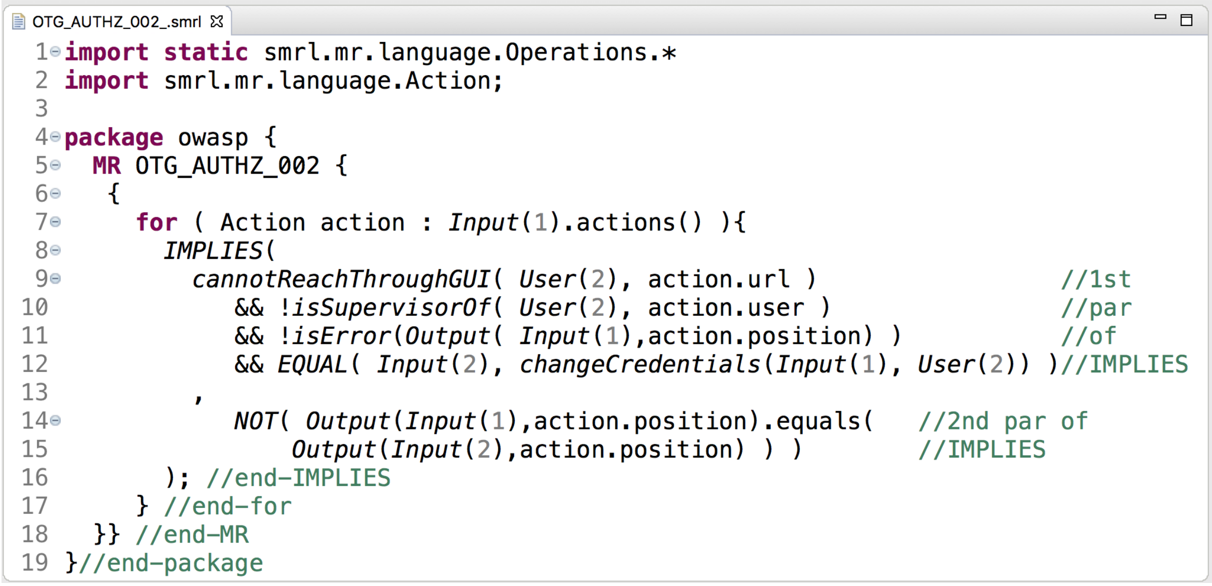
Figure 6.1. Overview of the approach.

addition, the engineer can also specify new relations by using our DSL. Step 1 is manual. We discuss this step in Section 6.3. In Step 2, our approach automatically transforms the MRs into executable Java code (Section 6.4).

In Step 3, the engineer executes a Web crawler to automatically collect information about the system under test (e.g., the URLs that can be visited by an anonymous user). The crawler determines the structure of the system under test and the actions that trigger the generation of new content on a page. The collected information includes the source inputs for MST. To collect additional information, the engineer can process manually implemented test scripts, if available. Step 3 does not depend on other steps. We discuss Step 3 in Section 6.5.

In Step 4, our approach automatically loads the source inputs required by the MRs and generates follow-up inputs as described by the relation. After the source and follow-up inputs are executed, their execution results are checked according to the MRs. The details of the step are described in Section 6.6.

Our DSL and the data collection framework can be extended to support new language constructs and data collection methods. The MST framework can be extended to deal with input interfaces not supported yet (e.g., Silverlight plug-ins [Microsoft Corp.,]) and to load data collected by new data collection methods.



```
import static smrl.mr.language.Operations.*
import smrl.mr.language.Action;

package owasp {
  MR OTG_AUTHZ_002 {
   {
     for ( Action action : Input(1).actions() ){
       IMPLIES(
         cannotReachThroughGUI( User(2), action.url )                 //1st
            && !isSupervisorOf( User(2), action.user )                //par
            && !isError(Output( Input(1),action.position) )           //of
            && EQUAL( Input(2), changeCredentials(Input(1), User(2)) )//IMPLIES
         ,
            NOT( Output(Input(1),action.position).equals(   //2nd par of
                Output(Input(2),action.position) ) )        //IMPLIES
       ); //end-IMPLIES
     } //end-for
  }} //end-MR
}//end-package
```

Figure 6.2. An MR for the Bypass Authorization Schema vulnerability.

6.3 SMRL: A DSL for Metamorphic Relations

Our approach starts with the activity of selecting and specifying MRs (Step 1 in Fig. 6.1). To enable specifying new MRs, we provide a DSL called Security Metamorphic Relation Language (SMRL). Engineers can also select MRs for the system under test from the set of predefined MRs.

SMRL is an extension of Xbase [Efftinge et al., 2012], an expression language provided by Xtext [Xte, 2018]. Xbase specifications can be translated to Java programs and compiled into executable Java bytecode. We rely on Xbase since DSLs extending Xbase inherit the syntax of a Java-like expression language as well as language infrastructure components, including a parser, a linker, a compiler and an interpreter [Efftinge et al., 2012]. These features will facilitate the adoption of SMRL.

SMRL extends Xbase by introducing (1) a set of data representation functions, (2) a set of boolean operators to specify security properties, and (3) a set of Web-specific functions to express data properties and transform data. These functions can also be extended by defining new Java APIs to be invoked in MRs.

Fig. 6.2 presents an MR written in our SMRL editor. The relation checks whether the URLs dedicated to specific users can be accessed by other users through a direct request. We use it as a running example.

In the following, we introduce the SMRL grammar, the boolean operators, the data representation functions, and the Web-specific functions.

Table 6.1. Excerpt of the data functions in SMRL.

Data function	Description
Input(int i)	Returns the ith input sequence.
Action(int i)	Returns the ith input action.
Session(int i)	Returns the ith Web session.
User(int i)	Returns the ith user of the system.
Output(Input i)	Returns the sequence of outputs generated by Input *i*.
Output(Input i, int n)	Returns the output generated by the nth action of Input *i*.
HttpMethod()	Returns the name of an HTTP method (e.g., DELETE).
RandomFilePath()	Returns a file system path. We select paths of files in the Web system subfolder, ignoring images, and replacing symbolic links (e.g., 'plugins' is mapped to 'plugin' in Jenkins).
RandomValue(Type t)	Returns a random value of the given type.

6.3.1 SMRL Grammar

The SMRL grammar extends the Xbase grammar, which extends the Java grammar. Each SMRL specification can have an arbitrary number of import declarations which indicate the APIs to be used in MRs (Line 1 in Fig. 6.2).

A package declaration resembles the Java package structure and can contain one or more MRs. Line 4 in Fig. 6.2 declares the package *owasp*, which is is the package for our MRs. Like in Java, MRs defined in different SMRL specification files can belong to the same package.

An MR can contain an arbitrary number of `XBlock- Expressions`, which are nonterminal symbols defined in the Xbase grammar. An `XBlockExpression` can contain loops, function calls, operators, and other `XBlockExpressions`.

6.3.2 Data Representation Functions

SMRL provides 18 functions to represent different types of data (i.e., system inputs and outputs) in MRs. Data is typically represented by a keyword followed by an index number used to identify different data items. To keep SMRL simple, we represent data by using functions (hereafter *data functions*) with capitalized names (e.g., `Input(1)`). Table 6.1 presents a subset of the data functions in SMRL.

Each data function returns a data class instance. Fig. 6.3 presents the SMRL data model where all classes are subtypes of either `InputType` or `OutputType`. `InputType` represents input data that can be defined to trigger a certain system behavior. `InputSequence` represents a sequence of interactions between a user and the system under test and is consequently associated with `Action`. `Action` represents an activity performed by a user (e.g., requesting a URL). It carries information about actions such as a URL requested by an action and parameters in the URL query string. `Action` is associated with `Session`, which represents a user session in a Web application. `User` represents a system user.

A *source input* is an instance of `InputType` returned by one of the data functions; a *follow-up* input is an instance of `InputType` modified by means of a Web-specific function (see Section 6.3.4). For example, a source input might be a sequence of two HTTP requests for user login and user profile

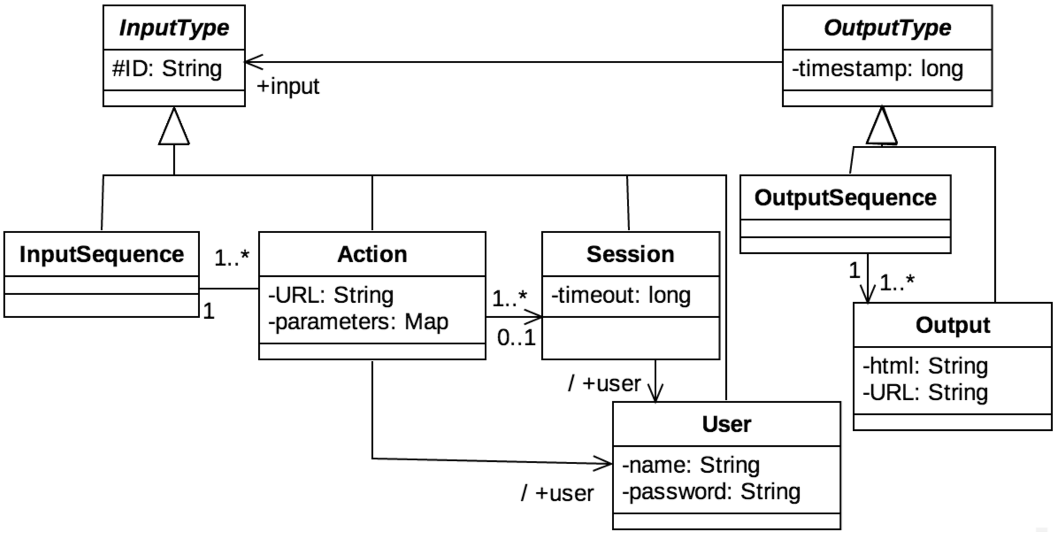


Figure 6.3. Metamorphic data classes in SMRL.

visualization. A follow-up input is the same sequence with login credentials for a different user. Instances of `OutputType` capture outputs generated by the system when processing an input; each instance of `OutputType` is associated with an instance of `InputType`. The last three functions in Table 6.1 return predefined/random values. They are used to redefine attributes of follow-up inputs as described in Section 6.6.

6.3.3 Boolean Operators

SMRL provides seven boolean operators, i.e., `IMPLIES`, `AND`, `OR`, `TRUE`, `FALSE`, `NOT` and `EQUAL`. They enable the definition of *metamorphic expressions*, which are boolean expressions that should hold for an MR to be true. A *metamorphic expressions* is a specific kind of `XBlockExpression`. We use metamorphic expressions to decompose an MR into simple properties. They are defined in a declarative manner, which is standard practice in MT.

The MR in Fig. 6.2 includes a metamorphic expression using the operator `IMPLIES`. Since the expression is within a loop body, the relation holds only if the expression evaluates to true in all the iterations over the input actions.

The semantics of the operators `IMPLIES`, `AND`, `OR`, `TRUE`, `FALSE`, and `NOT` is straightforward. The operator `EQUAL`, instead, does not simply evaluate the equality of two arguments but defines a follow-up input by assigning the second parameter to the first parameter. The operator `EQUAL` acts as an equality operator only when its first parameter refers to an input that has already been used in previous expressions of the MR. Otherwise, it acts as an assignment operator. In Fig. 6.2, the operator `EQUAL` defines the follow-up input `Input(2)` as a modified copy of `Input(1)`.

Table 6.2. Excerpt of the Web-specific functions in SMRL.

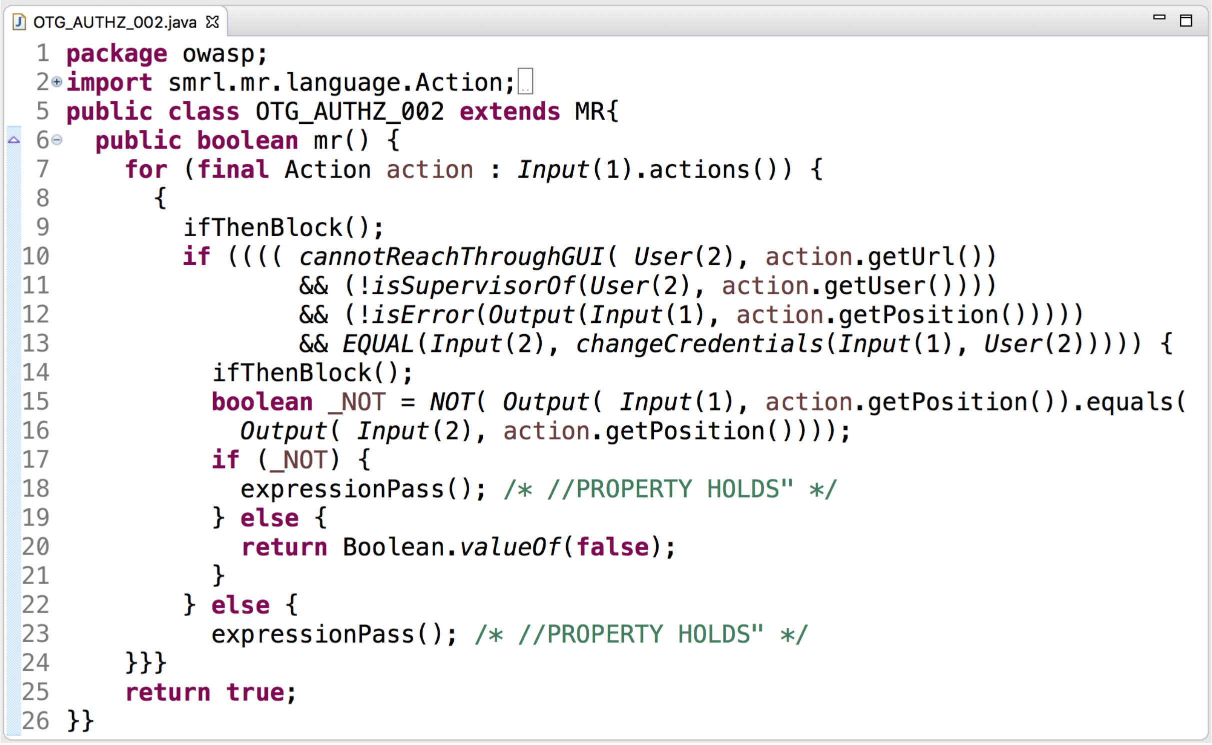
Operator	Description
changeCredentials(Input i, User u)	Creates a copy of the provided input sequence where the credentials of the specified user are used (e.g., within login actions).
copyActionTo(Input i, int from, int to)	Creates a new input sequence where an action is duplicated in the specified position and the remaining actions are shifted by one.
cannotReachThroughGUI(User u, String URL)	Returns true if a URL cannot be reached by the given user by exploring the user interface of the system (e.g., by traversing anchors).
isLogin(Action a)	Returns true if the action performs a login.
isSupervisorOf(User a,User b)	Returns true if 'a' can access the URLs of 'b'.
afterLogin(Action a)	Returns true if the action follows a login.
isSignup(Action a)	Returns true if the action registers a new user on the system.
isError(Output page)	Returns true if the page contains an error message.
userCanRetrieveContent(User u, Object out)	Returns true if the output data (i.e., the argument 'out') has ever been received in response to any of the input sequences executed by the given user during data collection.

6.3.4 Web-Specific Functions

MRs for security testing often capture complex properties of Web systems that cannot be expressed with simple boolean or arithmetic operators. Therefore, SMRL provides a set of functions that capture typical properties of Web systems and alter Web data. Table 6.2 describes a portion of the 30 Web-specific functions in SMRL [Web, 2019]. Each function is provided as a method of the SMRL API. Engineers can specify additional functions as Java methods. The new functions can be used in SMRL thanks to the underlying Xtext framework.

The MR in Fig. 6.2 uses the Web-specific functions `cannotReachThroughGUI`, `isSupervisorOf`, `isError` and `changeCredentials`. The relation indicates that the same sequence of actions should provide different outputs when performed by two different users under a certain condition. The condition is that one of the two users cannot access one of the requested URLs by simply browsing the GUI of the system. In other words, if the system does not provide a URL to a user through its GUI, then the user should not be allowed to access the URL. Also, to avoid false alarms, the user who cannot access the URL from the GUI, indicated as `User(2)` in Fig. 6.2, should not be a supervisor with access to all the resources of the other user, i.e., `User(1)`. Finally, we avoid source inputs that return an error message to `User(1)` because, for these inputs, it is not possible to characterize the output that should be observed for `User(2)`, who, indeed, may observe the same error, a different error, or an empty page.

In Fig. 6.2, the function `cannotReachThroughGUI` checks if the URL of the current action cannot be reached from the GUI (Line 9). The function `isSupervisorOf` checks if `User(2)` is not a supervisor of `User(1)` (Line 10). The function `isError` returns true if an output page contains an error message, based on a configurable regular expressions (Line 11). The function `changeCredentials` creates a copy of a provided input sequence using different credentials. It is invoked to define the follow-up input (Line 12). The data function `Output` executes the sequence of actions in an input sequence (e.g., requests a sequence of URLs) and returns the output of the i-th action.



OTG_AUTHZ_002.java

```
 1 package owasp;
 2 import smrl.mr.language.Action;
 5 public class OTG_AUTHZ_002 extends MR{
 6   public boolean mr() {
 7     for (final Action action : Input(1).actions()) {
 8       {
 9         ifThenBlock();
10         if (((( cannotReachThroughGUI( User(2), action.getUrl())
11                 && (!isSupervisorOf(User(2), action.getUser())))
12                 && (!isError(Output(Input(1), action.getPosition()))))
13                 && EQUAL(Input(2), changeCredentials(Input(1), User(2))))) {
14           ifThenBlock();
15           boolean _NOT = NOT( Output( Input(1), action.getPosition()).equals(
16             Output( Input(2), action.getPosition())));
17           if (_NOT) {
18             expressionPass(); /* //PROPERTY HOLDS" */
19           } else {
20             return Boolean.valueOf(false);
21           }
22         } else {
23           expressionPass(); /* //PROPERTY HOLDS" */
24     }}}
25     return true;
26 }}
```

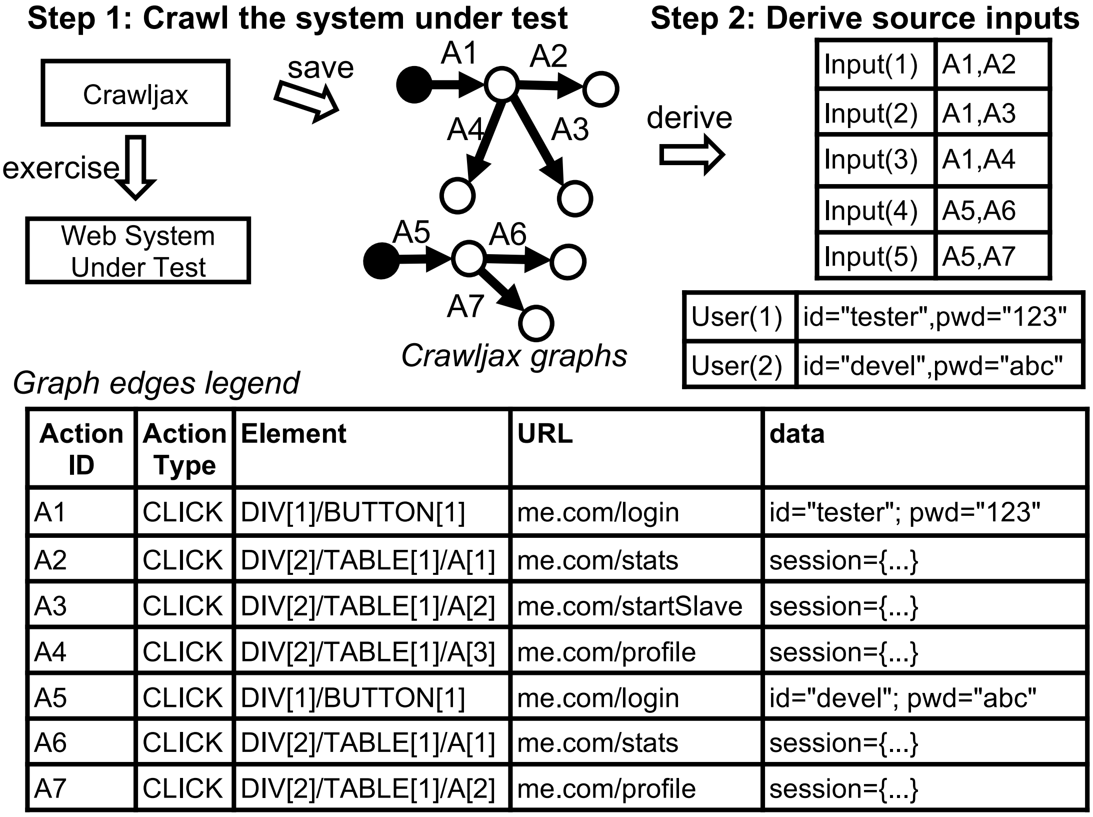
Figure 6.4. Java code generated from the MR in Fig. 6.2.

6.4 SMRL to Java transformation

SMRL specifications are automatically transformed into Java code (Step 2 in Fig. 6.1). To this end, we extended the Xbase compiler (hereafter SMRL compiler). Each MR is transformed into a Java class with the name of the relation and its package. The generated classes extend the class MR and implement its method mr.

The method mr executes the metamorphic expressions in the MR. It returns true if the relation holds and false otherwise. To do so, the SMRL compiler transforms each boolean operator into a set of nested IF conditions. For example, for the operator IMPLIES, the generated code returns false when the first parameter is true and the second one is false. For the case in which the MR holds, the SMRL compiler generates a statement that returns true at the end of mr.

Fig. 6.4 shows the Java code generated from the relation in Fig. 6.2. A loop control structure is generated from the loop instruction in the relation (Line 7). The loop body contains the Java code generated from the metamorphic expression using the operator IMPLIES (Lines 10-24). The first IF condition checks whether the first parameter of the operator IMPLIES holds (Lines 10-13). The nested IF block checks whether the second parameter of IMPLIES holds (Line 17). If the expression does not hold, mr returns false (Line 20). The relation holds only if all the expressions in the loop hold. Therefore, the SMRL compiler generates a return true statement after the loop body (Line 25). Calls to the methods ifThenBlock and expressionPass are used to erase the generated follow-up inputs at each iteration.



Input(1)	A1,A2
Input(2)	A1,A3
Input(3)	A1,A4
Input(4)	A5,A6
Input(5)	A5,A7

User(1)	id="tester",pwd="123"
User(2)	id="devel",pwd="abc"

Action ID	Action Type	Element	URL	data
A1	CLICK	DIV[1]/BUTTON[1]	me.com/login	id="tester"; pwd="123"
A2	CLICK	DIV[2]/TABLE[1]/A[1]	me.com/stats	session={...}
A3	CLICK	DIV[2]/TABLE[1]/A[2]	me.com/startSlave	session={...}
A4	CLICK	DIV[2]/TABLE[1]/A[3]	me.com/profile	session={...}
A5	CLICK	DIV[1]/BUTTON[1]	me.com/login	id="devel"; pwd="abc"
A6	CLICK	DIV[2]/TABLE[1]/A[1]	me.com/stats	session={...}
A7	CLICK	DIV[2]/TABLE[1]/A[2]	me.com/profile	session={...}

Figure 6.5. Data collection with a simplified example.

6.5 Data Collection Framework

To automatically derive source inputs (Step 3 in Fig. 6.1), we extended the Crawljax Web crawler [Mesbah et al., 2012, Mesbah et al., 2008]. Crawljax explores the user interface of a Web system (e.g., by requesting URLs in HTML anchors or by entering text in HTML forms). It generates a graph whose nodes represent the system states reached through the user interface and edges capture the action performed to reach a given state (e.g., clicking on a button). Crawljax detects states based on the content of the displayed page. Our extension relies on the edit distance to distinguish system states [Levenshtein, 1966]. We keep a cache of the HTML page associated to each state detected by Crawljax. When a new page is loaded, our extension computes the edit distance between the loaded page and all the pages associated to the different system states. When the distance is below a given threshold (5% of the page length), we assume that two pages belong to the same state. If a page does not belong to any state, Crawljax adds a new state to the graph. Crawling stops when no more states are encountered or a timeout is reached.

Our Crawljax extensions enable replicating and modifying portions of a crawling session. In addition to (i) the Crawljax actions and (ii) the XPath of the elements targeted by the actions (e.g., a button being clicked on), our extension records (iii) the URLs requested by the actions, (iv) the data in the HTML forms, and (v) the background URL requests. This enables, for example, replicating modified portions of crawling sessions that request URLs not appearing in the last Web page returned by the system. To crawl the system under test, we require only its URL and a list of credentials.

Fig. 6.5 exemplifies the data collection steps. First, Crawljax generates the graphs of the system under test. Second, source inputs are automatically derived from the graphs. For example, an input sequence is a path from the root to a leaf of a Crawljax graph in depth-first traversal. The source inputs are later queried by the SMRL functions (see Section 6.6). For example, `Input(i)` returns the i^{th} input sequence; `User(i)` returns the i^{th} unique login credentials in the input sequences.

In addition to Crawljax, our toolset also processes manually implemented test scripts to generate additional source inputs. It processes test scripts based on the Selenium framework [web, 2018] and derives a source input from each. We rely on test scripts to exercise complex interaction sequences not triggered by Crawljax (see Section 6.9). Crawljax, instead, performs an almost exhaustive exploration of the Web interface, which is typically not done by test scripts. Engineers can reuse scripts developed for functional testing, or define new ones.

6.6 Metamorphic Testing Framework

We automatically perform testing based on the executable MRs in Java and the data collected by the data collection framework (Step 4 in Fig. 6.1). Fig. 6.6 presents our testing algorithm. The algorithm takes as input a MR and a data provider exposing the collected data (source inputs). We first process the bytecode of the MR to identify the types of source inputs referenced by the relation (e.g., *Input* and *User*). This is achieved by the function `extractSourceInputTypes` (Line 2) which identifies the calls to the *data representation functions* using the ASM static analysis framework [ASM, 2018]. We ensure that all possible combinations of available source inputs are stressed during the execution of the relation (e.g., we would like to access all available URLs with all configured users). This is achieved by the function `iterateOverInputTypes` (Line 3). The function iterates over all available items for a given input type (e.g., all available users) and is recursively invoked for each input type in the MR.

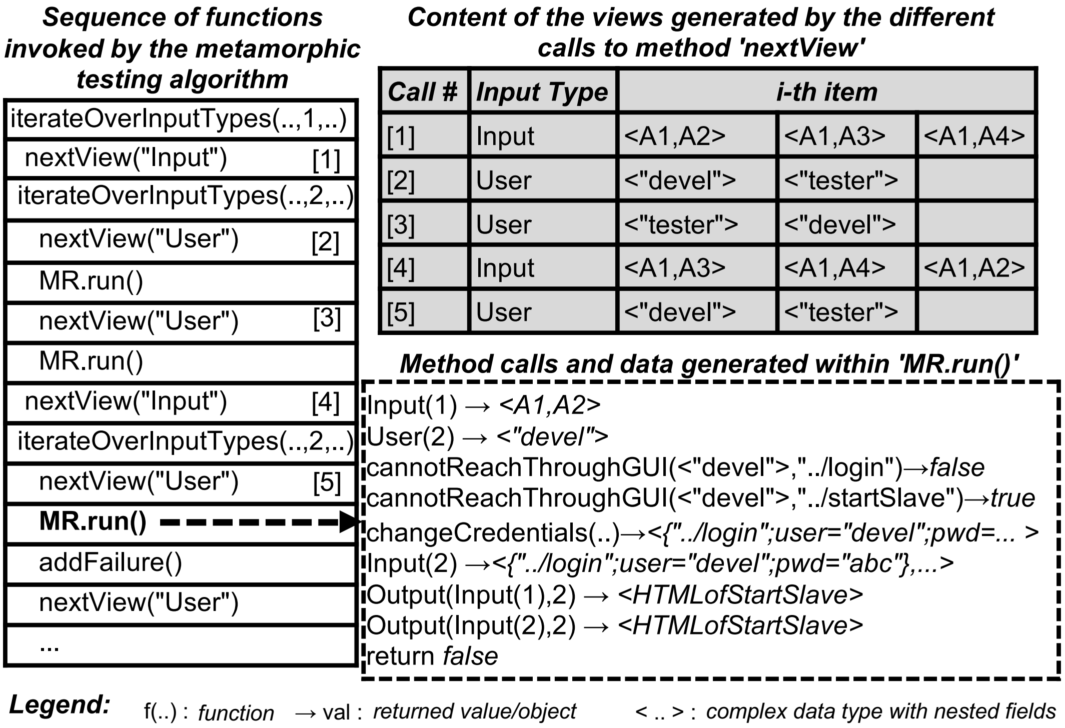
The function `iterateOverInputTypes` is driven by the methods exposed by the data provider (Lines 7 and 8). The data provider works as a circular array that provides, in each iteration of `iterateOverInputTypes`, a different view on the collected data. This is achieved through the method `nextView` (Line 8), which, for N input items of a given type (e.g., User), generates N different views, with items shifted by one position.

After the views are generated, the MR is executed (Line 12). Follow-up inputs are generated within the execution of the MR by the calls to the operator `EQUAL`. For example, in Fig. 6.2, the operator `EQUAL` makes `Input(2)` refer to a copy of the input sequence returned by the function `changeCredentials`.

When the relation does not hold (Lines 13 and 14), the function `addFailure` stores the failure context information (i.e., source-inputs, follow-up inputs, and system outputs). To minimize the time

```
Require: MR, the bytecode of the metamorphic relation to be executed
Require: dataProvider, an object that exposes the data collected by the crawlers
Ensure: Failures, a list of failing executions with contextual information
 1: function EXECUTEMETAMORPHICTESTING(MR, dataProvider)
 2:     srcTypes ← extractSourceInputTypes(MR)
 3:     iterateOverInputTypes(MR, dataProvider, 0, dataTypes)
 4:     return Failures
 5: end function
 6: function ITERATEOVERINPUTTYPES(MR, dataProvider, i, dataTypes)
 7:     while dataProvider.hasMoreViews(dataTypes[i]) do
 8:         dataProvider.nextView(dataTypes[i])
 9:         if (i < dataTypes.lenght) then   //need to iterate over other types
10:             iterateOverInputTypes(MR,dataProvider, i+1,srcTypes)
11:         else   //we have set a view for every input type in the relation
12:             result = MR.run() //execute the metamorphic relation
13:             if ( result == false) //the MR does not hold
14:                 addFailure(Failures,dataProvider) //trace the failure
15:         end if
16:     end while
17: end function
```

Figure 6.6. Metamorphic testing algorithm.


Sequence of functions invoked by the metamorphic testing algorithm
iterateOverInputTypes(..,1,..)
nextView("Input") [1]
iterateOverInputTypes(..,2,..)
nextView("User") [2]
MR.run()
nextView("User") [3]
MR.run()
nextView("Input") [4]
iterateOverInputTypes(..,2,..)
nextView("User") [5]
MR.run()
addFailure()
nextView("User")
...
Content of the views generated by the different calls to method 'nextView'
Method calls and data generated within 'MR.run()'
Legend: f(..) : function → val : returned value/object < .. > : complex data type with nested fields

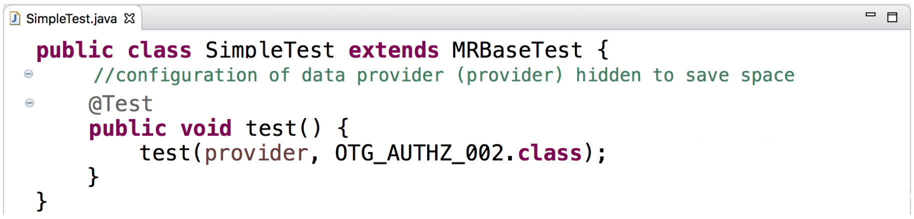
Call #	Input Type	i-th item		
[1]	Input	<A1,A2>	<A1,A3>	<A1,A4>
[2]	User	<"devel">	<"tester">	
[3]	User	<"tester">	<"devel">	
[4]	Input	<A1,A3>	<A1,A4>	<A1,A2>
[5]	User	<"devel">	<"tester">	

```
Input(1) → <A1,A2>
User(2) → <"devel">
cannotReachThroughGUI(<"devel">,"../login")→false
cannotReachThroughGUI(<"devel">,"../startSlave")→true
changeCredentials(..)→<{"../login";user="devel";pwd=... >
Input(2) →<{"../login";user="devel";pwd="abc"},...>
Output(Input(1),2) → <HTMLofStartSlave>
Output(Input(2),2) → <HTMLofStartSlave>
return false
```

Figure 6.7. Data processing for the relation in Fig. 6.2.

spent by engineers in analyzing failures triggered by distinct follow-up inputs exercising a same vulnerability, we report only failures that perform HTTP requests (e.g., accessing a URL) not generated by input sequences that led to previously reported failures.

Function `nextView` is iteratively invoked until all the items of a given input type are processed (Line 7). This guarantees that all input item combinations are used. For the data functions providing random values, `nextView` returns 100 different views by default. Since this may lead to combinatorial explosion, we test each MR for a maximum of 24 hours.



```
public class SimpleTest extends MRBaseTest {
    //configuration of data provider (provider) hidden to save space
    @Test
    public void test() {
        test(provider, OTG_AUTHZ_002.class);
    }
}
```

Figure 6.8. Example metamorphic test case.

Fig. 6.7 exemplifies the execution of the relation in Fig. 6.2. The table on the left represents the sequence of functions invoked by our algorithm. In this example, two views for `User` are inspected for each view of `Input`. The first two invocations of `MR.run` return true (not shown in Fig. 6.7) because the *login* and *stats* pages have been accessed by both users *devel* and *tester* and thus the implication holds. The third invocation of `MR.run` returns false because the output page for the *startSlave* URL is the same for the two input sequences and thus the relation does not hold. To determine if Web pages are equal, we rely on edit distance. Our framework relies on JUnit [JUn, 2017] to integrate MT into traditional testing environments (see Fig. 6.8). Engineers need only to configure the data provider and select the MR (s) to be tested.

6.7 Catalog of Metamorphic Relations

We derived a catalog of MRs from the activities described in the OWASP book on security testing [Meucci and Muller, 2014]. The book provides detailed descriptions of 90 testing activities (hereafter *OWASP testing activities*) for Web systems; each OWASP testing activity targets a specific vulnerability. For example, for the bypass authorization schema vulnerability, OWASP suggests to collect links in administrative interfaces and to directly access the corresponding URLs by using credentials of other users. Based on this suggestion, we defined the MR in Fig. 6.2.

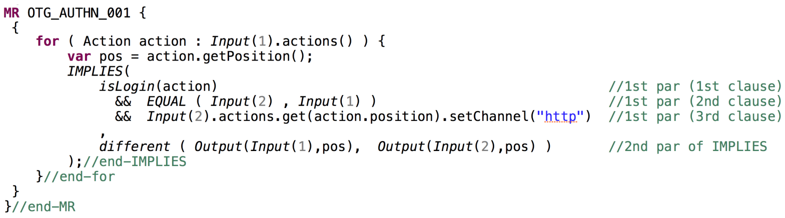
Some OWASP testing activities can be performed in multiple ways. Therefore, we have multiple relations for those activities. Also, not all the OWASP testing activities benefit from MT. The capabilities of MT are discussed in Section 6.9. We defined 22 MRs which automate 16 OWASP activities.

The MRs in our catalog rely on the observation that security testing might be performed using follow-up inputs that cannot be generated by interacting with the GUI of the system but conform with the input format of the system and match its configuration (e.g., the URLs requested by the unauthorized user refer to existing system resources). We inherit from mutational fuzzing the idea of generating follow-up inputs by altering valid source inputs. However, to generate inputs that are both valid and match the system configuration, instead of relying on random values, we alter source inputs using the data provided by the SMRL Web-specific functions, which return domain-specific information (e.g., protocol names) and crawled data. Finally, by capturing properties of the output generated by source and follow-up inputs we identify vulnerabilities that cannot be detected with implicit oracles.

Table 6.3. Excerpt of the metamorphic relation catalog for security testing.

<table>
<tr>
<td>

OTG-AUTHN-001: Testing for credentials transported over an encrypted channel



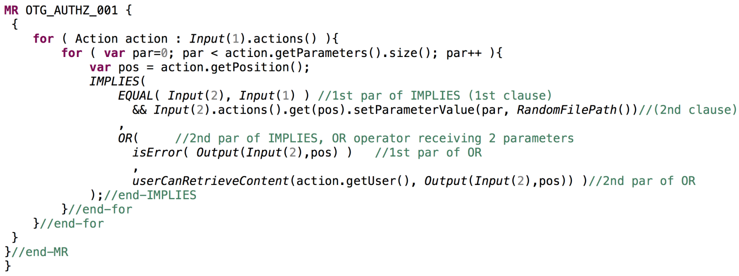
```
MR OTG_AUTHN_001 {
 {
    for ( Action action : Input(1).actions() ) {
        var pos = action.getPosition();
        IMPLIES(
            isLogin(action)                                              //1st par (1st clause)
              &&  EQUAL ( Input(2) , Input(1) )                          //1st par (2nd clause)
              &&  Input(2).actions.get(action.position).setChannel("http")  //1st par (3rd clause)
            ,
            different ( Output(Input(1),pos),  Output(Input(2),pos) )    //2nd par of IMPLIES
        );//end-IMPLIES
    }//end-for
 }
}//end-MR
```

</td>
<td>

Description: A login operation should not succeed if performed on the http channel. The 1st parameter of the operator `IMPLIES` is a boolean expression with three clauses joined with logical conjunctions. The 1st clause checks if the current action performs a login. The 2nd clause defines the follow-up input. The 3rd clause changes the channel of the login action in the follow-up input. The 2nd parameter of IMPLIES checks if the output generated by the login operation is different in the two cases.

</td>
</tr>
<tr>
<td>

OTG-AUTHZ-001: Testing for directory traversal/file include



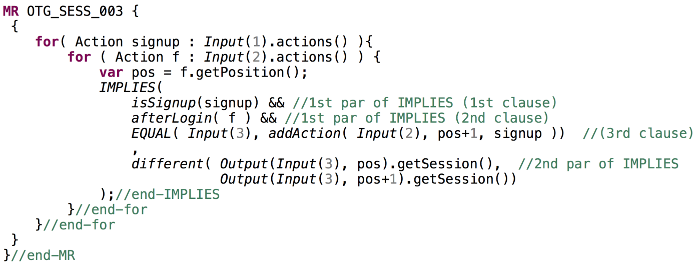
```
MR OTG_AUTHZ_001 {
 {
    for ( Action action : Input(1).actions() ){
        for ( var par=0; par < action.getParameters().size(); par++ ){
            var pos = action.getPosition();
            IMPLIES(
                EQUAL( Input(2), Input(1) ) //1st par of IMPLIES (1st clause)
                  && Input(2).actions().get(pos).setParameterValue(par, RandomFilePath())//(2nd clause)
                ,
                OR(     //2nd par of IMPLIES, OR operator receiving 2 parameters
                  isError( Output(Input(2),pos) )   //1st par of OR
                  ,
                  userCanRetrieveContent(action.getUser(), Output(Input(2),pos)) )//2nd par of OR
            );//end-IMPLIES
        }//end-for
    }//end-for
 }
}//end-MR
}
```

</td>
<td>

Description: A file path passed in a parameter should never enable a user to access data that is not provided by the user interface. This metamorphic relation contains two nested loops; the first iterates over the actions in the input sequence, the second iterates over the parameters of the action. The 1st parameter of the operator `IMPLIES` is a boolean expression with two clauses joined with a logical conjunction. The 1st clause defines a follow-up input that is a copy of the source input. The 2nd clause set the value of a parameter to a random file path. The 2nd parameter of IMPLIES verifies the result. It is implemented as an OR operation where the 1st parameter verifies that the follow-up input leads to an error page. The 2nd parameter deals with the case in which the generated request is valid, and verifies that the returned content is something that the user has the right to access. The framework evaluates the MR as many times as needed to provide 100 different random file paths to the parameters of the action in the position `pos`.

</td>
</tr>
<tr>
<td>

OTG-SESS-003: Testing for session fixation



```
MR OTG_SESS_003 {
 {
    for( Action signup : Input(1).actions() ){
        for ( Action f : Input(2).actions() ) {
            var pos = f.getPosition();
            IMPLIES(
                isSignup(signup) && //1st par of IMPLIES (1st clause)
                afterLogin( f ) && //1st par of IMPLIES (2nd clause)
                EQUAL( Input(3), addAction( Input(2), pos+1, signup ))  //(3rd clause)
                ,
                different( Output(Input(3), pos).getSession(),  //2nd par of IMPLIES
                           Output(Input(3), pos+1).getSession())
            );//end-IMPLIES
        }//end-for
    }//end-for
 }
}//end-MR
```

</td>
<td>

Description: A signup action should always lead to a new session ID, even when performed by a user who is already logged-in. This metamorphic relation contains two nested loops iterating over the actions of two distinct source input sequences (i.e., `Input(1)` and `Input(2)`). The first loop looks for a signup action (i.e., 'signup'), the second looks for an action (i.e., 'f') following a login. The 1st parameter of the operator `IMPLIES` is a boolean expression with three clauses joined with a logical conjunction. The 1st clause checks if we are in the presence of a signup action. The 2nd clause checks if the action 'f' follows a login. The 3rd clause defines a follow-up input by copying the signup action after the action 'f' in the source input `Input(2)`. The 2nd parameter of IMPLIES verifies the result by checking that the session ID following the signup action is different than the one of the previous page.

</td>
</tr>
</table>

Notes: Our catalog of metamorphic relations covers also the following OWASP activities: testing for HTTP Strict Transport Security (OTG-CONFIG-007), testing for weaker authentication in alternative channel (OTG-AUTHN-010), testing for privilege escalation (OTG-AUTHZ-003), testing for bypassing authentication schema (OTG-AUTHN-004), testing for insecure direct object references (OTG-AUTHZ-004), testing for logout functionality (OTG-SESS-006), test session timeout (OTG-SESS-007), testing for Session puzzling (OTG-SESS-008), testing for HTTP verb tampering (OTG-INPVAL-003), testing for HTTP parameter pollution (OTG-INPVAL-004), testing for weak encryption (OTG-CRYPST-004), test number of times a function can be used (OTG-BUSLOGIC-005), test for bypass authorization schema (OTG-AUTHZ-002, see Fig. 6.2).

Table 6.3 presents an excerpt of our catalog along with a description of each MR. The complete catalog of MRs is reported in Appendix B. All the MRs in the catalog are expressed by means of an implication (the operator IMPLIES). The operator EQUAL is used to define follow-up inputs. It indicates that the follow-up input (typically Input(2)) is a copy of the source input (usually Input(1)) except for the differences made by the function calls following the operator. For example, in OTG_AUTHN_001, the follow-up input is equal to the source input except for one action of the input sequence which should be performed on the HTTP channel.

All the MRs include a loop, which enables defining multiple follow-up inputs by iteratively modifying different actions of the source input. For example, OTG_AUTHN_001 works with all the login actions observed in the source input sequence. The function isLogin() returns true only if the current action performs a login; otherwise, the implication trivially holds and no follow-up input is generated.

In our catalog, the right-hand side of the implication usually captures the relation between the outputs of the source and follow-up inputs. In OTG_AUTHN_001, it is implied that the output for the follow-up input (which performs a login on the unencrypted HTTP channel) should be different than the output for the source input because it should not be possible to login using the HTTP channel.

6.8 Tool Support

The MST toolset supports our approach for security testing based on MRs. Fig. 6.9 provides an overview of our toolset. It consists of an Eclipse plugin, a library (i.e., SMRL.tar), and a Web crawler.

The *Eclipse plugin* provides the Editor for our SMRL language and automatically generates Java code from MRs (see Section 6.4). The Editor supports the auto-completion functionality to facilitate the specification of MRs. The SMRL compiler, which is extended from Xbase (see Section 6.4), automatically generates a Java class corresponding to each MR without any request to compile the MR.

The *MST library* (i.e., SMRL.tar) provides utility functions supporting the writing of MRs such as data representation functions (see Section 6.3.2), Boolean Operators (see Section 6.3.3), and Web-Specific functions (see Section 6.3.4). Moreover, our library supports functionalities of MST framework (see Section 6.6) such as the data provider, and MR, MRBaseTest classes.

The MST toolset relies on JUnit to automatically execute MT within Eclipse through our library. The Eclipse workspace is used to store all the data, which includes MRs and source inputs. Furthermore, the MST toolset includes the catalog of 22 system-agnostic MRs (sec Section 6.7).

The MST toolset, and usage instructions are available on the toolset's website at: **https://sntsvv.github.io/SMRL/**.

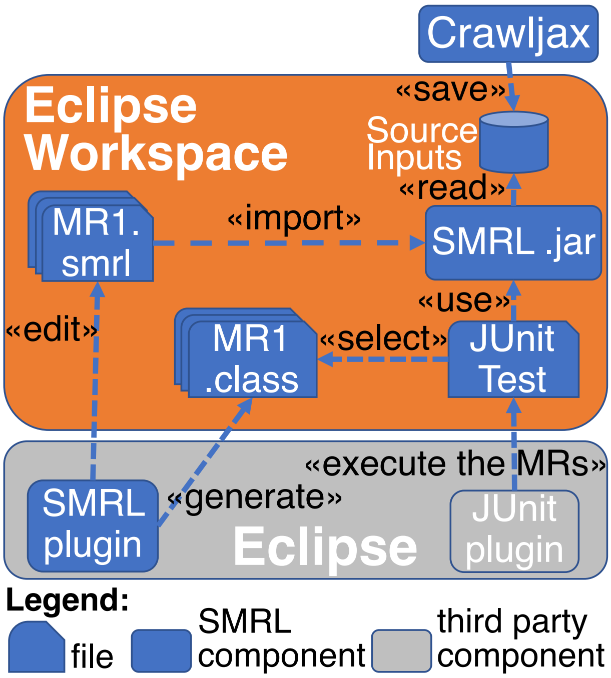


Figure 6.9. Components of the MST tool.

6.9 Evaluation

Our evaluation addresses the following research questions:

- ***RQ1. To what extent can metamorphic testing address the oracle problem in the context of security testing?*** We aim to determine which types of security vulnerabilities can be addressed by our solution.
- ***RQ2. Is the proposed solution effective?*** The goal is to assess whether the proposed solution enables, in a reliable manner, the automated detection of security vulnerabilities.

6.9.1 RQ1 - Targeted Types of Security Vulnerabilites

To answer RQ1, we analyzed the security testing activities recommended by OWASP [Meucci and Muller, 2014]. For each activity, we identified state-of-the-art oracle automation strategies. Table 6.4 lists the number of activities automated by these strategies. Details are available online [Web, 2019].

Implicit oracle. Some activities can be automated by random test input generation strategies relying on implicit oracles. For instance, *testing for buffer overflow* [OWASP, 2017d] is automated by looking for system crashes in response to lengthy inputs.

Catalog-based. We can automate some activities based on a predefined catalog in which we specify inputs and oracles. For instance, we can use a catalog to perform a dictionary attack for *testing for default credentials* [OWASP, 2017a].

Table 6.4. Oracle automation strategies for security testing

Oracle automation strategy	# OWASP activities automated
Implicit oracle	2
Catalog-based	6
No oracle needed	19
Manual oracle	25
Vulnerability-specific	22
Metamorphic testing	16

Table 6.5. Vulnerability types addressed by SMRL MRs

Vulnerability type	#MRs
Injection	0
Broken Authentication	6
Sensitive Data Exposure	5
XML External Entities(XEE)	0
Broken Access Control	7
Security Misconfiguration	3
Cross-site scripting (XSS)	0
Insecure Deserialization	0
Vulnerable Components	1
Insufficient Logging	0

No oracle needed. Some activities collect data to reverse engineer the system under test. They do not verify security properties of the system and thus do not have an oracle problem. For instance, the activity *mapping application architecture* [OWASP, 2017c] identifies the components of a Web system.

Manual oracle. Some activities require humans to determine vulnerabilities based on system specifications. For instance, when *testing for the circumvention of work flows* [OWASP, 2017b] on pay-per-view systems, only a human can decide if pending transactions should grant service access, based on specifications.

Vulnerability-specific approaches. Some activities can be automated by state-of-the-art tools such as Burp Suite (BS) [Portswigger, 2018a] and thus may not necessarily benefit from MT. These are the OWASP testing activities that detect cross site scripting and code injection vulnerabilities. Other activities are either not targeted or partially automated. For example, BS does not automate oracles for OTG-AUTHZ-002 [Portswigger, 2018c]. BS enables engineers to compare the content of site maps [Portswigger, 2018b] recorded in different user sessions (e.g., with and without certain privileges). Unfortunately, it requires that engineers manually identify the privileged resources and inspect the differences in the observed system outputs, which is error prone (e.g., overlooking privileged resources) and expensive. Even BS plug-ins using Crawljax to build site maps do not address the oracle problem but generate JUnit tests that simply retrieve the mapped resources [Liverani, 2018]. With SMRL, engineers, instead, can focus on the specification of system-level properties without performing manual testing activities. Testing activities, including oracles, are automated by the MT framework.

Metamorphic testing. All the other OWASP testing activities not addressed by the approaches above can be automated by MT. In general, these activities verify if a resource of the system under test can be accessed under circumstances that should prevent it (e.g., unauthenticated user or unencrypted channel). They benefit from MT since such activities entail the verification of all system resources, which are numerous and present specific security properties (e.g., each Web page might be accessed by a different set of users). For these activities, we provide a set of MRs (see Table 6.3).

Based on our analysis, out of 90 OWASP testing activities, 19 are not affected by the oracle problem, 30 are automated by state-of-the-art approaches, and 41 cannot be addressed by existing approaches. MT can automate 16 (39%) of these 41 activities. Therefore, *we conclude that MT can play a key role in addressing the oracle problem in security testing*.

To further characterize our catalog of MRs, we report in Table 6.5 the number of MRs targeting the vulnerability types in the OWASP top ten list [OWA, 2020b]. The MRs in our catalog can discover five of these ten vulnerability types, and thus *have a broad applicability scope*. Note that MRs can discover injection vulnerabilities [Huang et al., 2003] and, potentially, also XSS and XEE because they all concern injected code. In this chapter we specifically target vulnerabilities not addressed by existing oracle automation approaches, which is the reason why we ignored injections. We leave the investigation of other vulnerability types to future work.

6.9.2 RQ2 - Effectiveness

We applied the proposed approach MST to discover vulnerabilities in three case studies: the commercial Web system developed in the context of the EDLAH2 project [EDLAH2, 2017a], and two open source systems Jenkins [Eclipse Foundation, 2020] and Joomla [Joo, 2020]. As mentioned in Chapter 2, EDLAH2 is the entry point of a healthcare service developed by our industry partner [MiC, 2017]. The second case study, Jenkins, is an open-source continuous integration server. The third case study, Joomla, is an open-source content management system. We used the latest EDLAH2 version, Jenkins version 2.121.1, and Joomla version 3.8.7. EDLAH2 is affected by 12 vulnerabilities discovered by manual testing following the OWASP guidelines (See 5.8.1). We selected the Jenkins and Joomla versions affected by all the vulnerabilities triggerable from the Web interface, discovered in 2018, and reported in the Common Vulnerabilities and Exposures (CVE) database [MITRE Corporation, 2020] after June 1st, 2018[1]. Jenkins 2.121.1 and Joomla 3.8.7 are respectively affected by 20 and 16 such vulnerabilities.

Our approach addresses 36% (4 out of 11), 40% (8 out of 20), and 31% (5 out of 16) of the vulnerabilities affecting EDLAH2, Jenkins, and Joomla, respectively. This is consistent with our analysis in RQ1.

For each system under test, we configured our data collection framework with multiple users having different roles. We used two credentials for EDLAH2, four credentials for Jenkins, and six credentials for Joomla. For each role, we executed the data collection framework to crawl the system under test for a maximum of 300 minutes. In total, the data collection took 1000 minutes for Jenkins,

[1] after May 20, 2018 for the case of Joomla

Table 6.6. Summary of RQ2 results grouped by data collection method.

Case study	Vulnerabilities	Crawljax		Crawljax & Manual	
		Specificity	Sensitivity	Specificity	Sensitivity
EDLAH2	4 *	100.00%	75.00%	100.00%	100.00%
Jenkins 2.121.1	8 **	99.888%	50.00%	99.893%	75.00%
Joomla 3.8.7	2 ***	99.525%	50.00%	99.573%	100.00%
Overall	14	99.817%	57.14%	99.826%	85.71%

* See Section 5.8.1
** [MITRE, 2018h, MITRE, 2018g, MITRE, 2018f, MITRE, 2018e, MITRE, 2018d, MITRE, 2018c, MITRE, 2018b, MITRE, 2018a]
*** [MITRE, 2020a, MITRE, 2020b]

2280 minutes for Joomla, and 40 minutes for EDLAH2. For EDLAH2 and for the anonymous role in Jenkins and Joomla, Crawljax completed in less than 300 minutes because all states were visited. The data collection time for Joomla was long because Joomla has two different user interfaces (i.e., user and administrative interfaces). 73, 156, and 147 input sequences were identified for EDLAH2, Jenkins, and Joomla, respectively. Also, we implemented Selenium-based test scripts to exercise use cases not covered by Crawljax. This led to one, two, and one test scripts for EDLAH2, Jenkins, and Joomla, respectively. We tested the three systems against the MRs that target the vulnerabilities affecting them (4 for EDLAH2, 8 for Jenkins, and 2 for Joomla). In the case of Joomla, we considered only 2 out of the 5 vulnerabilities addressed by the approach because, due to the lack of detailed description of the attack scenarios, we could manually replicate only 2 out of 5 vulnerabilities. Our replicability package [Web, 2019] does not include EDLAH2 data because of confidentiality restrictions. Comparing with state-of-the-art tools is infeasible because they do not provide automated oracles.

We measured specificity and sensitivity [Lane, 2003]. Specificity (i.e., the true negative rate) is the ratio of follow-up inputs, generated by our framework, that do not trigger any vulnerability and (correctly) do not lead to any MT failure. In other words, *1 - specificity* measures the time spent by engineers on unwarranted MT failures. Sensitivity (i.e., the true positive rate) is the ratio of vulnerabilities being discovered. Based on the existing vulnerability reports for the two systems considered, we identified the inputs that should uncover vulnerabilities. MT failures are expected for these inputs to be true positives. For each MT failure, we manually verified if the test input actually triggered any vulnerability (true positive). Table 6.6 summarizes the results obtained with different data collection methods (i.e., based on Crawljax only or integrating Crawljax and manual test scripts). Each MR was tested in less than 12 hours, except five MRs run more than 24 hours. Performance optimizations are part of our future work.

We observe that the approach has **extremely high specificity** (99,826%), which indicates that only a negligible fraction of follow-up inputs inspected lead to false alarms (76 out of 43700, ~0.17%). False alarms are due to limitations in Crawljax, which, in cases of Jenkins and Joomla (i.e., 36 and 40 false alarms, respectively), did not traverse all the URLs provided by the GUI, for all the users. Consequently, MRs concerning authorization vulnerabilities fail. However, it is easy to determine that the URLs causing the false alarms should be accessible to all the users.

Sensitivity is high when data collection is based on both Crawljax and manual test scripts (100% for EDLAH2, 75% for Jenkins, and 100% for Joomla). Since sensitivity reflects the fault detection rate (i.e., the portion of vulnerabilities discovered), we conclude that our approach is **highly effective**. Overall, it detects 85.71% of the vulnerabilities targeted in our evaluation. More precisely, the approach identifies 146 distinct inputs sequences triggering these vulnerabilities. The approach misses two of the eight targeted vulnerabilities in Jenkins. One of them can be detected only if the server configuration is modified during test execution [MITRE, 2018f], which is not supported by our toolset. The other one cannot be reproduced since it concerns the termination of Jenkins' reboot [MITRE, 2018h], which is not interruptible when Jenkins is not overloaded (our case).

When the data collection relies on Crawljax only, sensitivity drops below 50% for both Jenkins and Joomla. This occurs since Jenkins requires quick system interactions to exercise certain features (e.g., first writing a valid Unix command in a textbox to enqueue a batch job, and then quickly pressing a button to delete it from the queue). For the case of Joomla, data collection is complicated by the presence of dynamic menus based on JavaScript. For instance, it might be necessary to first click on the menu *Components* to display the list of sub-menus, and then quickly click on the sub-menu *Tags*, which is unlikely performed by a Web crawler that clicks on randomly selected elements in the page. However, even when the data collection is based on Crawljax only, the overall fault detection rate is satisfactory (i.e., 57.14%), with 8 out of 14 vulnerabilities being detected. Automatically detecting 57.14% of the vulnerabilities not targeted by state-of-the-art approaches, without the need for any manual test script, is encouraging.

The benefits of our approach mostly stems from the MRs in our catalog being reusable to test any Web system. Furthermore, the required manual test scripts are few and inexpensive to implement. For the Web systems above, we manually wrote four test scripts which only include 17 actions in total. This is very limited in comparison to the total of 43700 inputs sequences (314907 actions) automatically generated by our approach to test the three systems. A traditional way to verify the same scenarios would require 43700 manually implemented test scripts, each providing a distinct input sequence, and a dedicated oracle (e.g., an assertion statement). Therefore, we conclude that our approach provides an advantageous cost-effectiveness trade-off compared to current practice.

Besides known vulnerabilities, our MST toolset assists in discovering a new vulnerability of Jenkins – Stored XSS vulnerability in file parameters. This new vulnerability of Jenkins enables a malicious user (with the 'job build' permission) to make administrators inadvertently perform privileged actions (i.e., delete an existing job in the system). This new vulnerability has been reported in the CVE vulnerability database with the identification CVE-2020-2162 [Sto, 2020].

6.9.3 Threats to Validity

The main threat to validity in our evaluation concerns the generalizability of the conclusions. Regarding RQ1, to mitigate this threat and minimize the risk of considering a set of testing activities that is not representative of Web application testing, we considered testing activities proposed by a third party organization (i.e., OWASP). As for RQ2, to mitigate this threat, we selected systems that are representative of modern Web systems but are very different from both a technical and process perspective.

6.10 Conclusion

In this chapter, we presented an approach that enables engineers to specify metamorphic relations (MR) capturing security properties of Web systems, and that automatically detects security vulnerabilities based on those relations. Our approach aims to alleviate the oracle problem in security testing.

Our contributions include (1) a DSL and supporting tools for specifying MRs for security testing, (2) a set of MRs inspired by OWASP guidelines, (3) a data collection framework crawling the system under test to automatically derive input data, and (4) a testing framework automatically performing security testing based on the MRs and the input data [Web, 2019].

Our analysis of the OWASP guidelines shows that MST can automate 39% of the security testing activities not currently targeted by state-of-the-art techniques, which indicates that the approach significantly contributes to addressing the oracle problem in security testing. Our empirical results with three commercial and open source case studies show that the MST approach requires limited manual effort and detects 85.7% of the targeted vulnerabilities, thus suggesting it is highly effective. Moreover, our MST approach helps to discover a new vulnerability.

Chapter 7

Applicability of Proposed Approaches and Testability Guidelines

In this chapter, we investigate which types of security vulnerabilities can be identified by the approaches proposed in this PhD dissertation (i.e.,MCP and MST), and what guidelines (hereafter, testability guidelines) should be followed to adopt these approaches in software projects.

We investigate the following Research Questions (RQs):

- ***RQ1. To what extent is MCP applicable in the context of security testing?*** MCP has been designed and implemented to test the user-system interactions involving some malicious activities. Not every type of vulnerability can be discovered through a sequence of interactions (e.g., some may require program analysis). This RQ aims to determine, in a systematic way, the types of security vulnerabilities that can and cannot be addressed by MCP.
- ***RQ2. Is it possible to define testability guidelines that enable effective test automation with MCP?*** Software testability is the degree to which a software artifact (i.e., a software system, module, requirements or design document) supports its testing [Voas and Miller, 1995]. A higher degree of testability results in decreased test effort, increased quality of test activities, a higher probability of findings software defects and, as a result, higher quality software. This RQ investigates if it is possible to identify testability guidelines that assist engineers in designing, implementing, and configuring their software to enable test automation with MCP.
- ***RQ3. To what extent is MST applicable in the context of security testing?*** MST is based on metamorphic relations that leverage source inputs that are sequences of inputs for the system under test that are either collected by means of a Web-crawler or encoded in manually implemented test scripts. In general, it may not be possible to define a metamorphic relation to determine if the system is vulnerable; also, some vulnerabilities may not be detected by simply altering the sequence of source inputs in a predefined manner but may be exploited only after identifying specific inputs by means of program analysis. For these reasons, the objective of this RQ is to determine the types of security vulnerabilities that can and cannot be addressed by MST.

- ***RQ4. Is it possible to define testability guidelines that enable test automation with MST?*** This RQ studies if it is possible to identify testability guidelines that assist engineers in designing, implementing, and configuring their software to enable test automation with MST.
- ***RQ5. How do MCP and MST compare in terms of applicability?*** This research question aims to investigate if the two approaches are complementary by discussing the type of vulnerabilities detected by both or by just one of them.
- ***RQ6. How do MCP and MST compare in terms of testability guidelines?*** This research question aims to investigate if the same testability guidelines enable applying both MCP and MST.

7.1 Subject of the Study

To address our research questions in a systematic way, we study the list of weaknesses reported in the Common Weakness Enumeration (CWE) database [CWE, 2020a].

We provide the following definitions of *vulnerability* and *weakness* since their definitions in the CWE framework [CWE, 2020l] lack clarity[1]. A *vulnerability* is a specific fault of the system under test that causes the system to not meet its security requirements. A *weakness* represents a fault type (i.e., the type of a vulnerability). It describes a human error made in the analysis, design, or implementation of the system that may affect the degree to which the system meets its security requirements.

The CWE database is organized into distinct views, each view grouping weaknesses according to a different set of categories, which are *common security architectural tactics* [CWE, 2020n], *software development concepts* [CWE, 2020q], *research concepts* [CWE, 2020p], *software fault patterns* [CWE, 2020k], *most dangerous errors* [CWE, 2020m], and *hardware design* [CWE, 2020o]. Other views map the weaknesses to some security-related catalogs (e.g., OWASP Top 10 [OWA, 2020a] and SERT CEI C Coding standards [CWE, 2020j]).

The CWE view for *common security architectural tactics* organizes weaknesses according to *security design principles*. This view has twelve categories representing the individual security design principles that are part of a secure-by-design approach to software development. It covers, in total, 223 weaknesses. The security design principles assist engineers in identifying potential mistakes that can be made when designing software [Santos et al., 2017a, Santos et al., 2017b]. A weakness is thus the result of a design principle not being followed. For instance, the weaknesses in design principle *Audit* are related to audit-based components in the system. These components deal with logging user activities to identify malicious users and modifications to the system [CWE, 2020n]. The views for

[1] The definitions provided by CWE are unclear since they rely on *synonyms* to distinguish vulnerability and weakness, as follows: 'Weaknesses are *flaws*, *faults*, *bugs*, and other *errors* in system design, architecture, code, or implementation that if left unaddressed could result in systems and networks, and hardware being vulnerable to attack. Weaknesses can lead to vulnerabilities. A vulnerability is a *mistake* in software or hardware that can be used by a malicious user to gain access to a system or network [CWE, 2020l]'.

Table 7.1. Subset of the security weaknesses in the CWE view for common security design principles and the applicability of MCP and MST.

Design principle	Weakness	CWE Top 25	OWASP Top 10	Generic	Addressed by		Testability Feature (TF) / Reason cannot apply (R)	
					MCP	MST	MCP	MST
Audit	Omission of Security-relevant Information	No	No	Yes	No	No	R2	R2
	Obscured Security-relevant Information by Alternate Name	No	No	Yes	Yes	No	TF1, TF2	R5
Authenticate Actors	Improper Authentication	Yes	Yes	Yes	Yes	Yes	TF3	TF3
	Weak Password Recovery Mechanism for Forgotten Password	No	No	Yes	No	No	R3	R3
Authorize Actors	Improper Privilege Management	Yes	No	Yes	Yes	Yes	TF3, TF7	TF3, TF7
	Process Control	No	No	Yes	No	No	R1	R1
Encrypt Data	Small Space of Random Values	No	No	Yes	No	No	R0, R2	R0, R2
	Unprotected Transport of Credentials	No	No	Yes	Yes	Yes	TF9	TF9
Limit Access	Improper Restriction of XML External Entity Reference	Yes	Yes	Yes	Yes	No	TF13	R6
	External Control of File Name or Path	Yes	No	Yes	Yes	Yes	TF3	TF3
Manage User Sessions	J2EE Bad Practices: Non-serializable Object Stored in Session	No	No	No	No	No	R1	R1
	Insufficient Session Expiration	No	No	Yes	Yes	Yes	TF4, TF11	TF4, TF11
Validate Inputs	Cross-site Scripting	Yes	Yes	Yes	Yes	No	TF13	R6
	Deserialization of Untrusted Data	Yes	Yes	Yes	No	No	R2	R2

software development concepts and *hardware design* organize weaknesses based on the types of errors that affect the software implementation (e.g., illegal pointer dereferences) and the hardware design (e.g., faults in semiconductor logic), respectively. The views for *software fault patterns* and *research concepts* group implementation errors into categories capturing fault patterns [Ben A. Calloni, 2011] or high level descriptions of the faulty software behaviour (e.g., incorrect comparison, or improper access control).

In our analysis, we focus on the weaknesses in the CWE view for common security architectural tactics [CWE, 2020n], the weaknesses in the view for the CWE Top 25 most dangerous software errors (CWE Top 25) [CWE, 2020m], and the weaknesses in the view for the OWASP Top 10 Web security risks (OWASP Top 10) [CWE, 2020r]. We select the common security architectural tactics view because it enables us to determine the security design principles that can be verified with MCP or MST. We do not consider the view for *software development concepts* because both MCP and MST are specifications-based (i.e., misuse case specification and metamorphic relations), black-box testing approaches, that do not aim to discover specific implementation errors (e.g., type errors). The specifications processed by MCP capture how a malicious user may behave, they are not inspired by the specific underlying implementation errors. For the same reason we ignore the views on *software fault patterns*, *research concepts*, and mappings to *coding standards* [CWE, 2020j] since they focus on software implementation. We also ignore the CWE view for *hardware design* since MCP and MST do not address hardware vulnerabilities.

However, we include the CWE Top 25 and OWASP Top 10 views to assess to what extent MCP and MST can address the most widespread and critical security vulnerabilities. The CWE Top 25 view lists twenty-five most widespread weaknesses, which are often easy to find and exploit. These weaknesses are considered dangerous because they often *allow attackers to completely take over the control of software, steal data, or prevent software from working [CWE, 2020m].* The OWASP Top 10 [OWA, 2020a] is the list of the ten most common web application security risks edited by the Open Web Application Security Project [OWASP, 2016], i.e., an online community producing freely-available articles, methodologies, documentation, tools, and technologies in the field of web application security. It is updated every three to four years. The most up-to-date version concerns 43 weaknesses grouped into 10 categories [CWE, 2020r].

To provide concrete examples of the weaknesses in our analysis, we report, in Table 7.1, a subset of the weaknesses in the CWE view for common security architectural tactics.We refer to Table 7.1 in the rest of this chapter. Columns *Design principle* and *Weakness* report the security design principle and the name of the weakness, respectively. Columns *CWE Top 25* and *OWASP Top 10* indicate whether a weakness also belongs to the CWE Top 25 view or the OWASP Top 10 view, respectively. We also indicate if the weakness can be addressed by MCP or MST. The remaining columns refer to concepts introduced later in this chapter.

7.2 RQ1: MCP Applicability

7.2.1 Measurements

To respond to RQ1, we compute, for each category in the views, the percentage of weaknesses that can be automatically discovered by MCP.

Weaknesses are systematically analyzed with the objective of writing, for each one, one or more misuse case specifications. For each weakness, we first inspect its description, its demonstrative examples, the description of the concrete vulnerabilities (CVE) and the common attack patterns (CAPEC) [CAP, 2020b] associated with the weakness. Common attack patterns are unstructured NL descriptions of the activities performed by malicious users to exploit a vulnerability. They resemble misuse case specifications, except that they are not written in a structured form that can be automatically analyzed by MCP. However, they facilitate the writing of misuse case specifications.

Based on the information collected from our inspection, we assess whether it is possible to write, using the RMCM template, a misuse case specification capturing a malicious user-system interaction. Each time we cannot do so, we keep track of the reasons preventing the writing of a specification.

We then report the percentage of the weaknesses that can be automatically discovered by MCP. Since some of the weaknesses in the CWE database are specific to certain types of systems (e.g., Java Enterprise [J2E, 2020]), we identify weaknesses that refer to specific systems. We then distinguish between the results achieved with all the weaknesses (i.e., generic and specific), and the results achieved with the generic weaknesses only. In Table 7.1, weakness *J2EE Bad Practices: Non-serializable Object Stored in Session* is an example of a system-specific weakness.

To better characterize the weaknesses that cannot be discovered by means of MCP, we analyze the distribution of the reasons why MCP cannot be applied, across the categories of the views considered in our analysis. Finally, we discuss the percentage of the weaknesses belonging to the CWE Top 25 and OWASP Top 10 weaknesses lists.

7.2.2 Results

Table 7.2 presents the summary of the CWE Security Design Principles and related security weaknesses addressed by MCP. The first column in Table 7.2 lists the security design principles appearing in the common security architectural tactics view. The second and third columns give, for each design principle, the overall number of weaknesses and the number of generic weaknesses, respectively. The fourth and fifth columns report the number and percentage of weaknesses that can be automatically discovered by MCP. In total, 131 out of all 223 weaknesses (59%) and 103 out of 164 generic weaknesses (63%) in the view can be addressed by MCP. These numbers show that MCP enables engineers to automatically discover a large portion of weaknesses. In addition, the higher percentage achieved for generic weaknesses shows that it performs better in the general case. Readers can download the details of our analysis for the 223 weaknesses from the MCP page [MCP, 2018].

MCP can automatically discover a high percentage of weaknesses (above 80%) related to security design principles *Lock Computer*, *Validate Inputs*, *Audit*, and *Identify Actors* (i.e., 100%, 85%, 83%, and 83%, respectively). These weaknesses are about external systems or actors with invalid certification trying to access the system (*Identify Actors*), their logging activities (*Audit*), providing malformed input data (e.g., code injection) to the system (*Validate Inputs*), or performing multiple attempts to access a given resource (*Lock Computer*). On the other hand, MCP addresses a low percentage of the weaknesses (below 20%) related to security design principles *Limit Exposure* and *Encrypt Data* (i.e., 0% and 18%, respectively). Our approach MCP is mainly used to test user-system interactions involving malicious activities. The weaknesses related to *Limit Exposure* and *Encrypt Data* are, on the contrary, about the quality of data encryption and the information that the system exposes, which require either static or dynamic program analysis with a human assessing the analysis outputs.

The three security design principles associated with the highest number of weaknesses are *Authorize Actors* (38 addressed out of 60 weaknesses - 63%), *Validate Inputs* (33 addressed out of 39 weaknesses - 85%), and *Encrypt Data* (7 addressed out of 38 weaknesses - 18%) . Only in the latter case, MCP addresses a low percentage of weaknesses.

Table 7.3 gives the summary of the CWE Top 25 weaknesses addressed by MCP. Our approach can automatically discover 15 out of the 25 top weaknesses (60%) and 14 out of the 23 generic top weaknesses (61%). MCP addresses all the top 25 weaknesses except the ones that require code analysis to be detected, i.e., *Improper Restriction of Operations within the Bounds of a Memory Buffer*, *Out-of-bounds Read*, *Use After Free*, *Out-of-bounds Write*, *NULL Pointer Dereference*, *Use*

Table 7.2. Summary of the CWE architectural security design principles and weaknesses addressed by MCP.

Security Design Principle	**weaknesses**		**addressed weaknesses**	
	all	**generic**	**all**	**generic**
Audit	6	6	5 (83%)	5 (83%)
Authenticate Actors	28	20	20 (71%)	15 (75%)
Authorize Actors	60	47	38 (63%)	31 (66%)
Cross Cutting	9	8	5 (56%)	4 (50%)
Encrypt Data	38	22	7 (18%)	6 (27%)
Identify Actors	12	8	10 (83%)	6 (75%)
Limit Access	8	7	5 (63%)	5 (71%)
Limit Exposure	6	4	0 (0%)	0 (0%)
Lock Computer	1	1	1 (100%)	1 (100%)
Manage User Sessions	6	3	4 (67%)	3 (100%)
Validate Inputs	39	28	33 (85%)	24 (86%)
Verify Message Integrity	10	10	3 (30%)	3 (30%)
Total	223	164	131 (59%)	103 (63%)

Table 7.3. Summary of the CWE Top 25 weaknesses addressed by MCP.

Weaknesses		**Addressed weaknesses**	
all	**generic**	**all**	**generic**
25	23	15 (60%)	14 (61%)

Table 7.4. Summary of the Security Weaknesses for OWASP Top 10 security risks addressed by MCP.

OWASP Security Risk	**Weaknesses**		**Addressed weaknesses**	
	all	**generic**	**all**	**generic**
Injection	9	8	8 (89%)	8 (100%)
Broken Authentication	9	8	6 (67%)	5 (63%)
Sensitive Data Exposure	11	10	6 (55%)	5 (50%)
XML External Entities	2	2	1 (50%)	1 (50%)
Broken Access Control	5	5	5 (100%)	5 (100%)
Security Misconfiguration	3	2	2 (67%)	1 (50%)
Cross-Site Scripting (XSS)	1	1	1 (100%)	1 (100%)
Insecure Deserialization	1	1	0 (0%)	0 (0%)
Using Components with Known Vulnerabilities	-	-	-	-
Insufficient Logging & Monitoring	2	2	1 (50%)	1 (50%)
Total	43	39	30 (70%)	27 (69%)

of Hard-coded Credentials, *Uncontrolled Resource Consumption*, *Missing Release of Resource after Effective Lifetime*, *Untrusted Search Path*, *Deserialization of Untrusted Data*.

For example, MCP can automatically verify weakness *Cross-site Scripting (XSS)* with a misuse case specification describing a Stored-XSS [CAP, 2020a] [CWE, 2020i] attack in which a malicious user uploads harmful scripts to the Web server, while another user (e.g., a system administrator) executes actions that run these harmful scripts [Sto, 2020].

Table 7.4 presents the summary of the security weaknesses related to the OWASP Top 10 security risks addressed by MCP. MCP can address 30 out of the 43 weaknesses (70%) in this view. Similar results are observed for the 39 generic weaknesses (69%). MCP addresses a high percentage (above 80%) of the weaknesses leading to security risks *Broken Access Control*, *Cross-Site Scripting*, and *Injection* (i.e., 100%, 100%, and 89%, respectively). These risks are about unauthorized access to

Table 7.5. Reasons preventing the application of MCP.

ID	**Reason**
R0	The weakness cannot be discovered by means of user-system interactions.
R1	The weakness concerns a system that is not Web-based or mobile-based.
R2	The weakness can be discovered only by means of program analysis.
R3	It is not possible to specify a regular expression to parse the system output and determine its correctness; a human needs to inspect it.

Table 7.6. Distribution of reasons preventing the application of MCP to discover Security Design Principles weaknesses.

Security Design Principle	**Weaknesses not addressed**	**R0**	**R1**	**R2**	**R3**	**Sum**
Audit	1			1		1
Authenticate Actor	8	1		6	1	8
Authorize Actor	22		**15**	6	1	22
Cross Cutting	4			3	1	4
Encrypt Data	31	**12**		**22**	1	35
Identify Actors	2		2			2
Limit Access	3		1		2	3
Limit Exposure	6			4	**6**	10
Lock Computer	0					0
Manage User Sessions	2		1		1	2
Validate Inputs	6		1	5		6
Verify Message Integrity	7			7		7
Total	92	13	20	**54**	13	100

resources, injecting malicious client-side scripts into a website, and sending invalid data to a web application. All involve malicious user-system interactions.

However, MCP addresses less than 50% of the weaknesses only for security risk *Insecure Deserialization*, which, however, includes only one weakness. For security risk *Using Components with Known Vulnerabilities*, there is no associated weakness in the CWE database. Since "known vulnerabilities" may arise from any kind of weakness, it is not possible to map this OWASP security risk to any specific weakness in the CWE database [OWA, 2017a].

Table 7.5 presents the reasons why MCP cannot be applied to discover weaknesses. MCP has been designed to generate test cases simulating the interaction between malicious users and Web-based or mobile-based systems. Therefore, MCP cannot be applied when the attack is not based on user-system interactions (see R0 in Table 7.5) or when the weakness concerns a system that is not Web-based or mobile-based (see R1 in Table 7.5). Also, some weaknesses can be discovered only by means of program analysis, which MCP does not support (see R2 in Table 7.5). In that case, the analysis should be either performed without actually executing programs (e.g., through code inspection) or the output of program analysis should be reviewed manually to eliminate false positives, which typically come in large numbers. R2 differs from R0. Indeed, R2 concerns weaknesses that, without system execution, can be proved to be exploitable by an attacker. Also, it is associated with weaknesses that require the system under test to be in a vulnerable state that is expensive to reach during testing. R0

Table 7.7. Distribution of preventing the application of MCP to discover weaknesses associated with the OWASP Top 10 security risks.

OWASP Security Risk	Weaknesses not addressed	R0	R1	R2	R3	Sum
Injection	1				1	1
Broken Authentication	3	1		1	1	3
Sensitive Data Exposure	5			5		5
XML External Entities	1			1		1
Broken Access Control	0					0
Security Misconfiguration	1				1	1
Cross-Site Scripting	0					0
Insecure Deserialization	1			1		1
Using Components with Known Vulnerabilities	-	-	-	-	-	-
Insufficient Logging & Monitoring	1			1		1
Total	13	1	0	**9**	3	13

Table 7.8. Distribution of reasons preventing the application of MCP to discover CWE Top 25 weaknesses.

Weakness	R0	R1	R2	R3
Improper Restriction of Operations within the Bounds of a Memory Buffer			1	
Out-of-bounds Read			1	
Use After Free			1	
Out-of-bounds Write			1	
NULL Pointer Dereference			1	
Use of Hard-coded Credentials			1	
Uncontrolled Resource Consumption			1	
Missing Release of Resource after Effective Lifetime				1
Untrusted Search Path		1		
Deserialization of Untrusted Data			1	
Total	0	1	**8**	1

concerns weaknesses that can be discovered only through system execution using a complex input difficult to identify at test specification time.

For some weaknesses, it is not always possible to specify a regular expression that determines if the system output matches a predefined pattern; such expressions are used as oracles to determine the correctness of the test output (see R3 in Table 7.5). For instance, weakness *Weak Password Recovery Mechanism for Forgotten Password* in Table 7.1 is a generic weakness that MCP cannot discover. This weakness indicates that the system under test contains a mechanism for the recovery of passwords that is weak (e.g., it is based on a security question whose answer can be easily determined [CWE, 2020h]). It is not possible to specify a pattern that recognizes an easy security question. An expert needs to manually inspect such question (see R3 in Table 7.5). Weakness *Omission of Security-relevant Information* in Table 7.1 is another weakness that cannot be discovered by MCP. It indicates that the system under test does not record information that would be important for identifying the source or nature of an attack, or for determining if an action is safe [CWE, 2020c]. For example, the system may log failed login attempts when a certain limit is reached. This can be verified by static program analysis techniques (see R2 in Table 7.5).

Table 7.6 presents the distribution of reasons preventing the application of MCP cannot be applied for the weaknesses regarding common security architectural tactics. Please note that there is more than one reason for some of the weaknesses (e.g., weakness *Small Space of Random Values* in Table 7.1). In 54 out of 92 weaknesses (59%) that cannot be discovered by MCP, program analysis is required (see R2). The security design principle with most of the weaknesses not being addressed by MCP is *Encrypt Data*, in which 12 out of 13 weaknesses (92%) are not addressed due to R0. Indeed, these weaknesses are related to the inefficiency of the random numbers generator (e.g., small range of random values and insufficient entropy) which could not be detected by triggering a sequence of user-system interactions, but requires a statistical analysis of the generated random values. Also, 22 out of 54 weaknesses (41%) are not addressed due to R2; indeed, these 22 weaknesses are associated with the protection of credentials or password (e.g., hard-coded cryptography key, password in configuration file), or the use of encryption/hash algorithm (e.g., weak cryptography, hash without a salt). Therefore, in these cases, a static program analysis approach should be used; for example, to find a hard-coded cryptography keys, or the invocation of weak encryption APIs.

The second design principle with most of the weaknesses not being addressed by MCP is *Authorize Actors*, for which MCP cannot address 22 weaknesses. 15 out of these 22 weaknesses (68%) concern a system that is not Web-based (R1). Indeed, these weaknesses affect the file system (e.g., preservation of permissions related to files, ownership management) or the process control in an operation system (e.g., Linux). These weaknesses cannot be discovered by an automated test framework dedicated to Web-based interactions, but require an analysis of the server configuration.

Security design principles *Encrypt Data* and *Authorize Actors* include the largest number of weaknesses not being addressed because of reasons R0, R1, and R2. Reason R3 affects mostly security design principle *Limit Exposure*, which indicates that the system should not provide information about its internal architecture or configuration (e.g., in error messages). To determine if this is an issue, it is necessary to inspect the system output. However, since it may come in many different forms, it is seldom possible to specify a regular expression that captures if information about the system internals has been provided.

Tables 7.7 and 7.8 report the reasons why MCP cannot be applied to discover some highly critical vulnerabilities. In these cases as well, the main reason is the necessity to rely on static program analysis to discover such vulnerabilities. This is expected since testing and program analysis are complementary quality assurance activities.

Summary. As a result of our analysis, we conclude that MCP can address most of the weaknesses organized in the CWE view for common security architectural tactics. It can also address a high percentage of high-risk weaknesses (60% of the CWE Top 25 weaknesses and 70% of the weaknesses related to the OWASP Top 10 security risks). These results are promising as they demonstrate that MCP is relevant for a large subset of vulnerabilities occurring in practice. Since MCP has been designed to test web-based or mobile-based systems for malicious user-system interaction, the weaknesses it cannot address are mostly those that can be discovered only by means of program analysis,

Table 7.9. Testability features and factors

ID	Testability feature	Testability factor
TF1	The system under test provides a feature to access log files	Observability
TF2	It is possible to define a test oracle by means of a regular expression to parse the system output	Observability
TF3	The feature under test is accessible via a URL/path	Controllability
TF4	The testing framework supports modifying parameter values	Test support environment
TF5	It is possible to log-in with a predefined list of credentials	Controllability
TF6	System settings or configuration elements can be controlled by the test engineer	Controllability
TF7	The testing framework can control the Web-browser (e.g., click on back button)	Test support environment
TF8	The type of the parameters of the request (in URL or post-data) is known or can be easily determined	Controllability
TF9	Parameters and sent data can be observed through a proxy	Observability
TF10	The testing framework provides data analysis functions	Test support environment
TF11	The system under test provides a feature to configure the system time	Controllability
TF12	The system under test provides access to intermediate results (e.g., hashed passwords)	Observability
TF13	The testing framework supports handling multiple user sessions in parallel	Test support environment
TF14	The testing framework has a feature to validate certificates	Test support environment

Table 7.10. Distribution of testability features for MCP.

Security Design Principle	Testability Feature Number													
	TF1	TF2	TF3	TF4	TF5	TF6	TF7	TF8	TF9	TF10	TF11	TF12	TF13	TF14
Audit	5	2	-	-	-	-	-	-	-	3	-	-	-	-
Authenticate Actors	-	-	3	8	5	-	-	-	1	-	2	1	-	-
Authorize Actors	1	3	26	10	-	3	3	1	1	-	-	-	-	-
Cross Cutting	-	-	2	3	-	-	-	-	-	-	-	-	-	-
Encrypt Data	2	-	1	1	-	-	-	-	5	-	-	-	-	-
Identify Actors	-	-	2	2	-	-	-	-	-	-	-	-	1	7
Limit Access	-	1	2	-	-	1	-	-	2	-	-	-	2	-
Limit Exposure	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Lock Computer	-	-	-	-	1	-	-	-	-	-	-	-	-	-
Manage User Sessions	-	-	1	1	-	-	-	-	-	-	1	-	2	-
Validate Inputs	1	-	19	19	-	-	-	-	1	-	-	-	2	-
Verify Message Integrity	-	-	2	2	-	-	-	-	-	-	-	-	-	-
Total	9	6	58	46	6	4	3	1	10	3	3	1	7	7
% of weaknesses *	7%	5%	45%	35%	5%	3%	2%	1%	8%	2%	2%	1%	5%	5%

* The percentage of weaknesses that can be discovered thanks to the testability feature.

that are not based on user-system interactions, or that concern a different type of systems than those targeted.

7.3 RQ2: Testability Guidelines for Applying MCP

7.3.1 Measurements

This research question investigates the possibility to define testability guidelines that support engineers in automatically testing software systems with MCP. More precisely, we aim to identify a set of features (hereafter, *testability features*) that should be provided either by the software under test or by the test framework and environment. Testability guidelines should indicate which testability features are required to detect specific categories of weaknesses, e.g., targeting a security design principle or entailing a high risk.

To identify testability features, we study the weaknesses that can be discovered by MCP, in the CWE view for common security architectural tactics. For each weakness, we identify a set of features that are necessary to enable automated testing with MCP.

The identification of features or, more generally, factors that affect or influence software testability is the subject of active research on testability. A recent survey [Garousi et al., 2019] lists 21 testability factors: *observability*, *controllability*, *complexity*, *cohesion*, *understandability*, *inheritance*, *reliability*, *availability*, *flexibility*, *test suite reusability*, *maintainability*, *unit size*, *statefulness*, *isolateability*, *software process capability*, *modularity*, *test support environment*, *fault-proneness*, *manageability*, *quality of the test suite*, and *self-documentation*. To guide engineers towards the inspection of the proposed testability features for MCP, we match each testability feature to the testability factors in the literature. This allows us to group related testability features.

Finally, we analyze the distribution of the testability features across the security design principles of the CWE view for common security architectural tactics, the security risks in the OWASP Top 10 CWE view, and the weaknesses in the CWE Top 25 view. This analysis should assist engineers in prioritizing the implementation of testability features for the system under test, based on the targeted weaknesses and security design principles.

7.3.2 Results

Table 7.9 presents the testability features for MCP and the corresponding testability factors. In total, we have 14 testability features. Nine features concern the system under test, while the other five features concern the test environment (see testability factor *Test support environment*).

In the case of MCP, three out of the 21 testability factors in the literature are required; these are (i) *Controllability* (i.e., the degree to which it is possible to control the state of the component under test [Garousi et al., 2019]), (ii) *Observability* (i.e., how easy it is to observe the behavior of a program in terms of its outputs, effects on the environment, and other hardware and software components [Garousi et al., 2019]), and (iii) *Test Support Environment*. In our context, testability factor *Test Support Environment* refers to the capability of the testing environment or framework to provide features to analyze system outputs or to alter the inputs transmitted to the system under test. The required testability factors are mostly determined by the type of testing performed by MCP: security vulnerability testing at system level by mimicking the actions performed by a malicious users. Therefore, to determine if the output of the system is correct, MCP may require improved *Observability*. To test the system under specific configurations, it needs high *Controllability*, and to automate activities typically performed by malicious users manually, the *Test Support Environment* requires a high degree of automation

Before discussing the distribution of testability features across design principles, we explain some of the testability features for the weaknesses in Table 7.1. For instance, weakness *Improper Authentication* is a generic weakness associated with design principle *Authenticate Actors*. It indicates that the system under test does not properly verify the identity claimed by an actor [CWE, 2020e]. MCP can be applied to identify this weakness when the feature under test is accessible through a URL/path (see TF3 in Table 7.9). We match this testability feature to testability factor *controllability*.

Another weakness that MCP can address is *Obscured Security-relevant Information by Alternate Name* [CWE, 2020d] associated with design principle *Audit*. In the weakness, the system records security-relevant information according to an alternate name of the file accessed by the malicious user but does not indicate that the two file names point to the same inode. Malicious users can hide their access to a file by using the alternate name of the file. A test case targeting this weakness checks whether only the alternate name of the affected file appears in the log file without any reference to the original file. To do so, we need a regular expression that distinguishes the malicious user's activity information (i.e., accessing a file) in log files. Therefore, MCP can be applied to identify this weakness when the system under test provides a feature to access log files (TF1), and when a regular expression can be used as an oracle (TF2), which requires proper log message templates. These testability features belong to testability factor *Observability*.

In weakness *Insufficient Session Expiration* [CWE, 2020g] in Table 7.1, a Web system permits malicious users to reuse old session credentials or session IDs for authorization. MCP automatically identifies this weakness only when it is possible to modify parameter values (TF4 concerning *Test Support Environment*), and when the system under test provides a feature to configure the system time (TF11 concerning *Controllability*). For instance, MCP can generate test cases that modify the HTTP-request (e.g., session IDs and cookie values) to reuse old session credentials. This is a feature implemented in the MCP test driver API.

Table 7.10 presents the distribution of the testability features for the security design principles in the CWE view for common security architectural tactics. Please note that there are more than one testability feature for some of the weaknesses associated with the security design principles. Indeed, we identify 164 testability features for 131 weaknesses concerning the 12 security design principles in Table 7.10. An example is weakness *Improper Privilege Management* in Table 7.1. Security design principles *Authorize Actors* and *Validate Inputs* are the ones that require the most testability features; this mostly depends on the fact that they are related the largest subset of weaknesses (see Table 7.2). In Table 7.10, rows *Total* and *% of weaknesses* show that the two testability features with the largest number of associated weaknesses are TF3 and TF4 (45% and 35%, respectively). These two features are related to testability factor *Controllability*.

In our analysis, we observe that *controllability*, *test support environment* and *observability* are required to address 53%, 51% and 15% of the weaknesses, respectively. These numbers are not fully in line with the literature on the topic, where the two most popular factors are observability (mentioned in 101 papers) and controllability (82 papers) [Garousi et al., 2019]. We believe that this difference is due to the fact that MCP performs security testing at the system level while the literature on testability mostly concerns functional and robustness testing.

Since MCP automates security testing by mimicking the actions performed by malicious users, it does not require a high degree of observability; indeed, the information that cannot be observed by

Table 7.11. Distribution of testability features of MCP for the weaknesses associated with the OWASP Top 10 security risks.

OWASP Security Risk	Testability Feature Number													
	TF1	**TF2**	**TF3**	**TF4**	**TF5**	**TF6**	**TF7**	**TF8**	**TF9**	**TF10**	**TF11**	**TF12**	**TF13**	**TF14**
Injection	-	-	5	5	-	-	-	-	-	-	-	-	-	-
Broken Authentication	-	-	2	1	1	-	-	-	1	-	1	-	1	-
Sensitive Data Exposure	2	1	1	-	-	-	-	-	3	-	-	-	-	1
XML External Entities	-	-	-	-	-	-	-	-	-	-	-	-	1	-
Broken Access Control	-	-	4	1	-	-	1	-	-	-	-	-	-	-
Security Misconfiguration	-	-	2	1	-	-	-	-	-	-	-	-	-	-
Cross-Site Scripting	-	-	-	-	-	-	-	-	-	-	-	-	1	-
Insecure Deserialization	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Using Components with Known Vulnerabilities	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Insufficient Logging & Monitoring	1	-	-	-	-	-	-	-	-	1	-	-	-	-
Total	3	1	14	8	1	0	1	0	4	1	1	0	3	1
% of weaknesses *	10%	3%	47%	27%	3%	0%	3%	0%	13%	3%	3%	0%	10%	3%

* The percentage of weaknesses that can be discovered thanks to the testability feature.

MCP cannot be observed also by malicious users, which indicates that such data is secure. The high relevance of testability factor *Test Support Environment* for MCP is due to the fact that, to mimic a malicious user, it is necessary to automate all the actions typically performed manually by attackers.

Tables 7.11 and 7.12 show the testability features that enable testing for the weaknesses in the OWASP Top 10 and CWE Top 25 views, respectively. Numbers are in line with the ones in Table 7.10. Indeed, the testability features that are required for testing a higher subset of weaknesses are also TF3 and TF4 in Tables 7.11 and 7.12. Similar trends are observed for TF9, TF13, and TF14, which address more than 5% of the weaknesses in Tables 7.10, 7.11, and 7.12.

Tables 7.10, 7.11, and 7.12 provide testability guidelines for engineers. They enable engineers to determine, based on the security requirements of the system under test, which testability features need to be enabled. For example, if the system provides authorization and authentication features, engineers need to implement the security design principles *Authenticate Actors* and *Authorize Actors*. Consequently, it might be useful to ensure that these two features under test are accessible through a URL/path (TF3) and that the testing framework supports modifying parameter values (TF4). Also, more generally, TF3 and TF4 enable test automation for highly critical weaknesses, which concern code injection, authorization, and authentication. In addition, by looking at the testability features addressing more than 5% of the vulnerabilities in Tables 7.10, 7.11, and 7.12 (i.e., at least two weaknesses associated with the OWASP Top 10 risks, one weakness in the CWE Top 25 list, and five weaknesses concerning the Security Design Principles), it is possible to identify a minimal set of testability features (i.e., TF1, TF3, TF4, TF9, TF13, and TF14) that should be prioritized to automatically verify both Security Design Principles and Top Security Risks. Finally, if the system under test does not need to identify actors through certificates, it is not necessary to rely on a testing framework with a feature to validate certificates (TF14).

Table 7.12. Distribution of testability features of MCP for the CWE Top 25 weaknesses.

CWE Top 25 Weakness	**Testability Feature Number**													
	TF1	**TF2**	**TF3**	**TF4**	**TF5**	**TF6**	**TF7**	**TF8**	**TF9**	**TF10**	**TF11**	**TF12**	**TF13**	**TF14**
Cross-Site Scripting	-	-	-	-	-	-	-	-	-	-	-	-	1	-
Improper Input Validation	-	-	1	1	-	-	-	-	-	-	-	-	-	-
Information Exposure	-	1	-	-	-	-	-	-	1	-	-	-	-	-
SQL Injection	-	-	1	1	-	-	-	-	-	-	-	-	-	-
Integer Overflow or Wraparound	-	-	-	1	-	-	-	-	-	-	-	-	-	-
Cross-Site Request Forgery (CSRF)	-	-	-	-	-	-	-	-	-	-	-	-	1	-
Path Traversal	-	-	1	-	-	-	-	-	-	-	-	-	-	-
OS Command Injection	-	-	1	-	-	-	-	-	-	-	-	-	-	-
Improper Authentication	-	-	1	-	-	-	-	-	-	-	-	-	-	-
Incorrect Permission Assignment for for Critical Resource	-	-	1	-	-	-	-	-	-	-	-	-	-	-
Unrestricted Upload of File with Dangerous Type	-	-	-	1	-	-	-	-	-	-	-	-	-	-
Improper Restriction of XML External Entity Reference	-	-	-	-	-	-	-	-	-	-	-	-	1	-
Code Injection	-	-	-	1	-	-	-	-	-	-	-	-	-	-
Improper Privilege Management	-	-	1	-	-	-	1	-	-	-	-	-	-	-
Improper Certificate Validation	-	-	-	-	-	-	-	-	-	-	-	-	-	1
Total	0	1	7	5	0	0	1	0	1	0	0	0	3	1
% of weaknesses *	0%	7%	47%	33%	0%	0%	7%	0%	7%	0%	0%	0%	20%	7%

* The percentage of weaknesses that can be discovered thanks to the testability feature.

7.4 RQ3: MST Applicability

7.4.1 Measurements

Following the same procedure adopted for RQ1, to address RQ3, we analyze the weaknesses in the CWE views *Security Design Principles*, *CWE Top 25*, and *OWASP Top 10* with the objective of specifying, for each weakness, one or more MRs. For each weakness, we first inspect its description, its demonstrative examples, the description of the concrete vulnerabilities (CVE) and the common attack patterns (CAPEC) [CAP, 2020b] associated with the weakness. Based on the information collected from our inspection, we assess whether it is possible to use an available MR in our catalog of MRs (see Section 6.7) or to specify a new MR to address these weaknesses. Each time we cannot specify a MR for a weakness, we keep track of the reasons for not doing so.

To address RQ3, we discuss the weaknesses that can be automatically discovered by MST and we study the distribution of the reasons preventing the application of MST, across the categories of the views considered in our analysis.

7.4.2 Results

Table 7.13 presents the summary of the CWE Security Design Principles and related security weaknesses addressed by MST. The first column in Table 7.13 presents the security design principles appearing in the CWE view for common security architectural tactics. The second and third columns give, for each design principle, the overall number of weaknesses and the number of generic weaknesses, respectively. The fourth and fifth columns report the number and percentage of weaknesses that can be automatically discovered by MST. In total, 101 out of all 223 weaknesses (45%) and 78 out of 164 generic weaknesses (48%) in the view can be addressed by MST. These numbers show that MST enables engineers to automatically discover a large portion of weaknesses. This is consistent with our conclusion in Section 6.9.1, where we state that MST can automate 39% of OWASP testing activities (each activity targets a different type of vulnerability). In addition, the higher percentage achieved for generic weaknesses shows that it performs slightly better in the general case. The details of our analysis for the 223 weaknesses is reported in Appendix C and on the MST Web page [Web, 2019].

MST can automatically discover a high percentage of weaknesses (above 60%) related to security design principles *Identify Actors* (75%), *Manage User Sessions* (67%), and *Validate Inputs* (67%). Weaknesses related to the principle *Identify Actors* concern malicious users trying to access the system with invalid certificates. Weaknesses related to the principle *Manage User Sessions* are about resources accessed by malicious users because of session management faults. Weaknesses related to *Validate Inputs* allow attackers to provide malformed input data (e.g., code injection) to the system, which enables attackers to affect either confidentiality, integrity, or accessibility. On the other hand, MST addresses a low percentage of the weaknesses (below 20%) related to security design principles *Audit*, *Limit Exposure*, *Lock Computer*, and *Encrypt Data* (i.e., 0%, 0%, 0% and 13%, respectively). This is mostly due to our MST being based on MRs that use as source inputs sequences of user-system interactions collected by a Web crawler, while these weaknesses concern mostly the analysis of data. Indeed, the weaknesses related to *Audit*, *Limit Exposure*, *Lock Computer*, and *Encrypt Data* are, on the contrary, about the quality of recorded logs, the information that the system exposes, restrictions of the lock out mechanism (e.g., lock an account after a predefined number of failed logins), and the quality of data encryption, all of them requiring the manual inspection of data.

The three security design principles associated with the highest number of weaknesses are *Authorize Actors* (32 addressed out of 60 weaknesses - 53%), *Encrypt Data* (5 addressed out of 38 weaknesses - 13%), and *Validate Inputs* (26 addressed out of 39 weaknesses - 67%). The exception is *Encrypt Data*, for which MST addresses a low percentage of weaknesses.

Table 7.14 provides a summary of the CWE Top 25 weaknesses addressed by MST. MST can automatically discover nine out of the CWE Top 25 weaknesses (36%) and nine out of the 23 (i.e., 39%) generic weaknesses belonging to the Top 25 view. These percentages are in line with our analysis in Section 6.9.1 (i.e., MST addressing 39% of vulnerabilities). For example, MST can automatically verify the weakness *Improper Input Validation* with a MR including follow-up inputs containing either malformed input data (e.g., date, time, and special characters), or equivalent inputs

Table 7.13. Summary of the CWE architectural security design principles and weaknesses addressed by MST.

Security Design Principle	weaknesses		addressed weaknesses	
	all	generic	all	generic
Audit	6	6	0 (0%)	0 (0%)
Authenticate Actors	28	20	15 (54%)	11 (55%)
Authorize Actors	60	47	32 (53%)	25 (53%)
Cross Cutting	9	8	5 (56%)	4 (50%)
Encrypt Data	38	22	5 (13%)	5 (23%)
Identify Actors	12	8	9 (75%)	5 (63%)
Limit Access	8	7	3 (38%)	3 (43%)
Limit Exposure	6	4	0 (0%)	0 (0%)
Lock Computer	1	1	0 (0%)	0 (0%)
Manage User Sessions	6	3	4 (67%)	3 (100%)
Validate Inputs	39	28	26 (67%)	20 (71%)
Verify Message Integrity	10	10	2 (20%)	2 (20%)
Total	223	164	101 (45%)	78 (48%)

Table 7.14. Summary of the CWE Top 25 weaknesses addressed by MST.

Weaknesses		Addressed weaknesses	
all	generic	all	generic
25	23	9 (36%)	9 (39%)

Table 7.15. Summary of the Security Weaknesses for OWASP Top 10 security risks addressed by MST.

OWASP Security Risk	Weaknesses		Addressed weaknesses	
	all	generic	all	generic
Injection	9	8	7 (78%)	7 (88%)
Broken Authentication	9	8	6 (67%)	5 (63%)
Sensitive Data Exposure	11	10	4 (37%)	3 (30%)
XML External Entities	2	2	0 (0%)	0 (0%)
Broken Access Control	5	5	5 (100%)	5 (100%)
Security Misconfiguration	3	2	2 (67%)	1 (50%)
Cross-Site Scripting (XSS)	1	1	0 (0%)	0 (0%)
Insecure Deserialization	1	1	0 (0%)	0 (0%)
Using Components with Known Vulnerabilities	-	-	-	-
Insufficient Logging & Monitoring	2	2	0 (0%)	0 (0%)
Total	43	39	24 (56%)	21 (54%)

(e.g., using both "-e" and "–exec", which are are the same command-line switch that could be used when calling an external program) [CWE, 2020b]. This MR helps to check if the system under test validates all input values. MST cannot address the top 25 weaknesses that require program analysis or interactions from third parties (e.g., other system users or system administrators) to be detected, i.e., *Improper Restriction of Operations within the Bounds of a Memory Buffer*, *Cross-site Scripting*, *Information Exposure*, *Out-of-bounds Read*, *Use After Free*, *Integer Overflow or Wraparound*, *Cross-Site Request Forgery (CSRF)*, *Out-of-bounds Write*, *NULL Pointer Dereference*, *Improper Restriction of XML External Entity Reference*, *Improper Control of Generation of Code ('Code Injection')*, *Use of Hard-coded Credentials*, *Uncontrolled Resource Consumption*, *Missing Release of Resource after Effective Lifetime*, *Untrusted Search Path*, and *Deserialization of Untrusted Data*.

Table 7.15 presents the summary of the security weaknesses related to the OWASP Top 10 se-

Table 7.16. Reasons preventing the application of MST.

ID	Reason
R0	The weakness cannot be discovered by means of user-system interactions.
R1	The weakness concerns a system that is not Web-based or mobile-based.
R2	The weakness can be discovered only by means of program analysis.
R3	It is not possible to distinguish valid and invalid behaviour based on system output; a human needs to inspect it.
R4	The weakness can be discovered only by means of data analysis.
R5	MT is inefficient. The weakness can be discovered only one test case execution.
R6	The weakness can be discovered only with results retrieved from a third side.

curity risks addressed by MST. MST can address 24 out of the 43 weaknesses (56%) and 21 out of 30 generic weaknesses (54%) in this view. MST addresses a high percentage (above 60%) of the weaknesses leading to security risks *Broken Access Control*, *Injection*, *Broken Authentication*, and *Security Misconfiguration* (i.e., 100%, 78%, 67%, and 67%, respectively). These risks are about unauthorized access to resources, injecting malicious client-side scripts into a website, bypassing the authentication, and gaining system information thanks to system misconfiguration. All involve malicious user-system interactions.

MST cannot address any of the weaknesses related to the four security risks *XML External Entities*, *Cross-Site Scripting (XSS)*, *Insecure Deserialization*, and *Insufficient Logging & Monitoring* which, however, include only a total of six weakness. For the security risk *Sensitive Data Exposure*, 37% (4 out of 11) weaknesses can be addressed by MST. This percentage is aligned with our study in the Section 6.9.1. For security risk *Using Components with Known Vulnerabilities*, there is no associated weakness in the CWE database (detail in Section 7.2).

In total, MST discovers 104 weaknesses from the three CWE views analyzed in this work. Among them, 101 weaknesses are related to the Security Design Principles view, and three weaknesses belong to the OWASP Top 10 view only. Among the discovered weaknesses, 21 belong to both the Security Design Principles view and the OWASP Top 10 view, while eight belong to both the Security Design Principles view and the CWE Top 25 view.

Table 7.16 presents the reasons why MST cannot be applied to discover weaknesses. The strategy currently adopted by MST, to generate source inputs, relies on either a Web crawler to automate the user-system interactions or manually implemented test scripts that automate such interactions. Such source inputs are turned into follow-up inputs which are used to discover vulnerabilities. Therefore, MST can discover vulnerabilities that can be exercised by means of a sequence of interactions and thus cannot be applied to discover weaknesses due to reasons R0, R1, and R2, similar to the case of MCP (see Section 7.2).

In some cases, it is not possible to define a MR because a human is needed to inspect the system output (R3). For instance, to discover the weakness *Weak Password Recovery Mechanism for Forgotten Password* (see Table 7.1), a human needs to indicate that the system under test contains a mechanism for the recovery of passwords that is weak (e.g., it is based on a security question whose

Table 7.17. Distribution of reasons preventing the application of MST to discover Security Design Principles weaknesses.

Security Design Principle	**Weaknesses not addressed**	**R0**	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**	**Sum**
Audit	6			1		**3**	2		6
Authenticate Actor	13	1		6	1		**4**	1	13
Authorize Actor	28		**16**	6	2	2	2		28
Cross Cutting	4			3	1				4
Encrypt Data	33	**12**		**22**	1		**4**		39
Identify Actors	3		2				1		3
Limit Access	5		1		2		1	1	5
Limit Exposure	6			4	**6**				10
Lock Computer	1						1		1
Manage User Sessions	2		1		1				2
Validate Inputs	13		1	5				**7**	13
Verify Message Integrity	8			7			1		8
Total	122	13	21	**54**	14	5	16	9	132

Table 7.18. Distribution of reasons preventing the application of MST to discover weaknesses associated with the OWASP Top 10 security risks.

OWASP Security Risk	**Weaknesses not addressed**	**R0**	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**	**Sum**
Injection	2			1	1				2
Broken Authentication	3	1		1	1				3
Sensitive Data Exposure	7			5		1	1		7
XML External Entities	2			1				1	2
Broken Access Control	0								0
Security Misconfiguration	1				1				1
Cross-Site Scripting	1							1	1
Insecure Deserialization	1			1					1
Using Components with Known Vulnerabilities	-	-	-	-	-	-	-	-	-
Insufficient Logging & Monitoring	2			1		1			2
Total	19	1	0	10	3	2	1	2	19

answer can be easily determined [CWE, 2020h]). Reason R3 is a generalization of reason R3 presented in Section 7.2.2, i.e., *It is not possible to specify a regular expression to parse the system output and determine its correctness; a human needs to inspect it.*

Because of the characteristic of MT (based on MRs), MST cannot be applied to address weaknesses that can be discovered only by means of data analysis (see R4 in Table 7.16). More precisely, these weaknesses can be found out only by analyzing large amounts of data (e.g., log files) based on statistics or machine learning; these types of analyses cannot be specified by means of a MR. In addition, for some weaknesses, it is not efficient to apply an MT approach because they can be discovered with a single test case execution (see R5 in Table 7.16). In that case, it is not needed to generate follow-up inputs from a large number of source inputs, which is the main characteristic of MT. Finally, some weaknesses can be discovered only with results retrieved from a third side (e.g., passwords stored on a third party server) or with interactions from a third side (e.g., a victim

Table 7.19. Distribution of reasons preventing the application of MST to discover CWE Top 25 weaknesses.

Weakness	**R0**	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**
Improper Restriction of Operations within the Bounds of a Memory Buffer			1				
Cross-site Scripting							1
Information Exposure				1			
Out-of-bounds Read			1				
Use After Free			1				
Integer Overflow or Wraparound				1			
Cross-Site Request Forgery (CSRF)							1
Out-of-bounds Write			1				
NULL Pointer Dereference			1				
Improper Restriction of XML External Entity Reference							1
Code Injection							1
Use of Hard-coded Credentials			1				
Uncontrolled Resource Consumption			1				
Missing Release of Resource after Effective Lifetime				1			
Untrusted Search Path		1					
Deserialization of Untrusted Data			1				
Total	0	1	**8**	3	0	0	**4**

clicks on the harmful URL provided by an attacker) (see R6 in Table 7.16). Expressing these types of interactions by means of metamorphic relations appears infeasible in the general case.

Please note that R0, R2, and R4 represent three distinct cases. R0 characterizes weaknesses that can be detected based on an input that cannot be derived by means of a predefined transformation function used in MRs. R2 concerns weaknesses that can be directly detected by means of program analysis. R4 concerns weaknesses that cannot be detected by verifying a single output but require the (statistical) analysis of multiple outputs.

Table 7.17 presents the distribution of reasons why MST cannot be applied for the weaknesses in the CWE view for common security architectural tactics. Please note that there is more than one reason for some of the weaknesses (e.g., weakness *Small Space of Random Values* in Table 7.1). In 54 out of 122 weaknesses (44%) that cannot be discovered by MST, the reasons lie in the necessity of using program analysis to discover the weaknesses (see R2 in Table 7.16), which is expected since program analysis complements software testing in software verification.

The security design principle with most of the weaknesses not being addressed by MST is *Encrypt Data*, with 12 out of 13 weaknesses (92%) not addressed because of R0. Indeed, these weaknesses are related to the inefficiency of a random numbers generator (e.g., small range of random values and insufficient entropy) which could not be detected by triggering a sequence of user-system interactions, but requires the statistical analysis of the generated random values. Also, 22 out of 54 weaknesses (41%) are not addressed because of R2; indeed, these 22 weaknesses are associated with the protection of credentials or password (e.g., hard-coded cryptography key, password in configuration files), or the use of encryption/hash algorithm (e.g., weak cryptography, hash without a salt). Therefore, in these cases, a static program analysis approach should be used; for example, to find a hard-coded cryptography key, or the invocation of weak encryption API. The four remaining weaknesses should not be detected by using MST because they require only one test case execution (see R5 in Table 7.16).

For example, to discover the weakness *Cleartext Storage of Sensitive Information in a Cookie* [CWE, 2020f], the test engineer only needs to log into the Web system under test one time and use a network sniffer or a proxy to capture the cookie. The test engineer then checks if the cookie contains a password in clear text. MST is inefficient to capture this type of weaknesses.

The second design principle with most of the weaknesses not being addressed by MST is *Authorize Actors*, for which MST cannot address 28 weaknesses. 16 out of these 28 weaknesses (57%) concern a system that is not Web-based (see R1). Indeed, these weaknesses affect the file system (e.g., preservation of permissions related to files, ownership management), the process control in an operation system (e.g., Linux), or the process communication in a mobile operation system (e.g., Android). These weaknesses cannot be discovered by an automated test framework dedicated to Web-based interactions, but require an analysis of the server/device configuration.

Security design principles *Encrypt Data* and *Authorize Actors* include the largest number of weaknesses not applied because of reasons R0, R1, and R2. Reason R3 affects mostly security design principle *Limit Exposure*, which indicates that the system should not provide information about its internal architecture or configuration (e.g., in error messages). To determine weaknesses concerning this design principle, it is necessary to inspect the system output. However, a MR cannot automatically capture (e.g., extract information from error messages) internal information of system. A human is usually needed to inspect the system output. Reason R4 affects mostly the security design principle *Audit* which requires to assess the quality of system logs (e.g., containing sensitive information, containing too much information, or too little information). To this end, we need data analysis (e.g., based on statistics or machine learning) to analyze system logs. Furthermore, it is not possible to specify a MR which describes the quality of system logs in a relation. Reason R5 typically affects weaknesses belonging to security design principles *Authenticate Actor* and *Encrypt Data*. Indeed, weaknesses concerning the security design principle *Authenticate Actor* affect the management of passwords (e.g., weak password requirements, not using password aging) which could be detected through a single test case execution. Reason R5 concerns 13% of the weaknesses (i.e., 16 out of 122). Reason R6 affects 7 out of 13 (54%) weaknesses not addressed by MST and concerning the security design principle *Validate Inputs*. These weaknesses are about injecting code into the Web system under test (e.g., *Cross-Site Scripting*, *Cross-Site Request Forgery*). In these cases, to determine if the attack can be performed, we require interactions from a third actor (e.g., an administrator, a valid user) who should inadvertently execute the injected code. Deriving interactions from a third actor using source inputs appears infeasible, at the current stage.

Tables 7.18 and 7.19 report the reasons why MST cannot be applied to discover some highly critical vulnerabilities. Even in these cases, the main reason is the necessity to rely on static program analysis (R2).

Summary. As a result of our analysis, we conclude that MST can address a significant number of the weaknesses listed in the CWE view for common security architectural tactics. It can also address a good percentage of high-risk weaknesses (56% of the weaknesses related to OWASP Top 10 security risks and 36% of CWE Top 25 weaknesses). We find the results of our analysis promising regarding MST's feasibility and widespread applicability. The weaknesses that MST cannot address are mostly those that (1) can be discovered only by means of program analysis, (2) are not based on user-system interactions, or (3) concern a non Web-based system. Also, MST is inefficient for weaknesses that could be discovered by means of a single test case execution.

7.5 RQ4: Testability Guidelines for Applying MST

7.5.1 Measurements

This research question evaluates the possibility to define testability guidelines that support engineers in automatically testing software systems with MST. More precisely, we aim to identify a set of *testability features* that should be provided either by the software under test or by the test framework and environment.

To identify testability features, we study the weaknesses that can be discovered by MST, in the CWE view for common security architectural tactics, the OWASP Top 10 CWE view, and the CWE Top 25 view. For each weakness, we identify a set of features that are necessary to enable automated testing with MST. Similar to the case of MCP (see Section 7.3), to guide engineers towards the inspection of the proposed testability features for MST, we match each testability feature to the testability factors in the literature. This enables us to group testability features based on the associated testability factors.

Finally, we analyze the distribution of the testability features across the security design principles of the CWE view for common security architectural tactics, the security risks in the OWASP Top 10 CWE view, and the weaknesses in the CWE Top 25 view. This analysis could assist engineers in prioritizing the implementation of testability features for the system under test, based on the targeted weaknesses and security design principles.

7.5.2 Results

Since MST triggers sequences of interactions with the system under test, it shares the same testability features described in Table 7.9 for MCP. In total, MST requires 10 out of the 14 testability features in Table 7.9. These 10 testability features belong to three testability factors (i.e., *Controllability*, *Observability*, and *Test Support Environment*). MST does not require *TF1*, *TF2*, *TF10*, and *TF12* (a discussion of the diferences between MST and MCP is reported in Section 7.7).

Table 7.20 presents the distribution of the testability features for the security design principles in the CWE view for common security architectural tactics. Please note that there might be more than one testability feature associated to each weakness. An example is the weakness *Insufficient Session Expiration*, which is associated to both TF4 and TF11 (see Table 7.1). Precisely, TF4 supports the test engineer to reuse an old session (e.g., credentials or ID) by modifying the corresponding request

Table 7.20. Distribution of testability features for MST.

Security Design Principle	**Testability Feature Number**													
	TF1	**TF2**	**TF3**	**TF4**	**TF5**	**TF6**	**TF7**	**TF8**	**TF9**	**TF10**	**TF11**	**TF12**	**TF13**	**TF14**
Audit	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Authenticate Actors	-	-	3	9	2	-	-	-	1	-	-	-	-	-
Authorize Actors	-	-	24	10	-	3	3	1	-	-	-	-	-	-
Cross Cutting	-	-	2	2	-	-	-	-	-	-	-	-	-	-
Encrypt Data	-	-	1	1	-	-	-	-	3	-	-	-	-	-
Identify Actors	-	-	2	2	-	-	-	-	-	-	-	-	-	7
Limit Access	-	-	2	-	-	1	-	-	1	-	-	-	1	-
Limit Exposure	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Lock Computer	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Manage User Sessions	-	-	1	1	-	-	-	-	-	-	1	-	2	-
Validate Inputs	-	-	17	15	-	-	-	-	-	-	-	-	-	-
Verify Message Integrity	-	-	1	2	-	-	-	-	-	-	-	-	-	-
Total	0	0	53	42	2	4	3	1	5	0	1	0	3	7
% of weaknesses *	0%	0%	52%	42%	2%	4%	3%	1%	5%	0%	1%	0%	3%	7%

* Percentage of weaknesses that can be discovered thanks to a testability feature.

parameters while TF11 helps to change the system time in order to invalid this session (i.e., expired). If the test engineer is authorized to access the system, the system is suffering the weakness *Insufficient Session Expiration*. In total, we identify 121 testability features for 101 weaknesses concerning the 12 security design principles in Table 7.20. Security design principles *Authorize Actors* and *Validate Inputs* are the ones that require the largest number of testability features; this mostly depends on the fact that they are related the largest portion of weaknesses (see Table 7.13). In Table 7.20, rows *Total* and *% of weaknesses* show that the two testability features with the largest number of associated weaknesses are TF3 and TF4 (52% and 42%, respectively). These two features are related to testability factor *Controllability*.

In our analysis, we observe that *Controllability*, *Test Support Environment* and *Observability* are required to address 57%, 54% and 5% of the weaknesses, respectively. Similar to the case of MCP, these numbers are not fully in line with the literature on the topic (see Section 7.3). MST automatically simulates actions performed by a user on the Web system under test under specific conditions (e.g., after performing a login). It thus requires a high degree of controllability to exercise the features under test or control the state of the system, instead of a high degree of observability. To automate all the actions that are typically executed manually by attackers, testability features concerning the *Test Support Environment* are required.

Tables 7.21 and 7.22 show the testability features that enable testing for the weaknesses in the OWASP Top 10 and CWE Top 25 views, respectively. Numbers are in line with the ones in Table 7.20. Indeed, also in Tables 7.21 and 7.22, the testability features that are required for testing a higher portion of weaknesses are TF3 and TF4. A similar trend across Tables 7.20, 7.21, and 7.22 is observed also for TF9, which addresses more than 5% of the weaknesses.

Tables 7.20, 7.21, and 7.22 provide testability guidelines for engineers. They enable engineers to determine, based on the security requirements of the system under test, which testability features need to be enabled. For example, if the system implements authorization and authentication features,

Table 7.21. Distribution of testability features of MST for the weaknesses associated with the OWASP Top 10 security risks.

OWASP Security Risk	**Testability Feature Number**													
	TF1	**TF2**	**TF3**	**TF4**	**TF5**	**TF6**	**TF7**	**TF8**	**TF9**	**TF10**	**TF11**	**TF12**	**TF13**	**TF14**
Injection	-	-	4	5	-	-	-	-	-	-	-	-	-	-
Broken Authentication	-	-	2	1	1	-	-	-	1	-	1	-	1	-
Sensitive Data Exposure	-	-	1	-	-	-	-	-	2	-	-	-	-	1
XML External Entities	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Broken Access Control	-	-	4	1	-	-	1	-	-	-	-	-	-	-
Security Misconfiguration	-	-	2	1	-	-	-	-	-	-	-	-	-	-
Cross-Site Scripting	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Insecure Deserialization	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Using Components with Known Vulnerabilities	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Insufficient Logging & Monitoring	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Total	0	0	13	8	1	0	1	0	3	0	1	0	1	1
% of weaknesses *	0%	0%	54%	33%	4%	0%	4%	0%	13%	0%	4%	0%	4%	4%

* Percentage of weaknesses that can be discovered thanks to a testability feature.

Table 7.22. Distribution of testability features of MST for the CWE Top 25 weaknesses.

CWE Top 25 Weakness	**Testability Feature Number**													
	TF1	**TF2**	**TF3**	**TF4**	**TF5**	**TF6**	**TF7**	**TF8**	**TF9**	**TF10**	**TF11**	**TF12**	**TF13**	**TF14**
Improper Input Validation	-	-	1	1	-	-	-	-	-	-	-	-	-	-
SQL Injection	-	-	1	1	-	-	-	-	-	-	-	-	-	-
Path Traversal	-	-	1	-	-	-	-	-	-	-	-	-	-	-
OS Command Injection	-	-	1	-	-	-	-	-	-	-	-	-	-	-
Improper Authentication	-	-	1	-	-	-	-	-	-	-	-	-	-	-
Incorrect Permission Assignment for for Critical Resource	-	-	1	-	-	-	-	-	-	-	-	-	-	-
Unrestricted Upload of File with Dangerous Type	-	-	-	1	-	-	-	-	-	-	-	-	-	-
Improper Privilege Management	-	-	1	-	-	-	1	-	-	-	-	-	-	-
Improper Certificate Validation	-	-	-	-	-	-	-	-	-	-	-	-	-	1
Total	0	0	7	3	0	0	1	0	1	0	0	0	0	1
% of weaknesses *	0%	0%	78%	33%	0%	0%	11%	0%	11%	0%	0%	0%	0%	11%

* The percentage of weaknesses that can be discovered thanks to the testability feature.

it might be useful to ensure that all the features under test are accessible through a URL/path (TF3) and that the testing framework supports modifying parameter values (TF4). Moreover, TF3 and TF4 also enable test automation for the highly critical weaknesses, which concern code injection, authentication, and authorization. In addition, by looking at the testability features addressing more than 5% of the vulnerabilities in Tables 7.20, 7.21, and 7.22 (i.e., at least five weaknesses concerning the Security Design Principles, two weaknesses associated with the OWASP Top 10 risks, and one weakness in the CWE Top 25 list), it is possible to identify a minimal set of testability features (i.e., TF3, TF4, and TF9) that is necessary to automatically verify both Security Design Principles and Top security risks. Since TF3, TF4, and TF9 are common in Web-oriented systems, we conclude that MST is likely applicable in most software projects without the need for adapting existing design or testing framework.

7.6 RQ5: Complementary Applicability of MCP and MST

7.6.1 Measurements

In RQ1 and RQ3, we discussed separately the applicability of MCP (see Section 7.2) and MST (see Section 7.4) to automate the testing of web-oriented software systems. In this Section, we aim to study if the two approaches are complementary by discussing the type of vulnerabilities discovered by both.

7.6.2 Results

Table 7.23 presents the numbers of weaknesses in Security Design Principles that can be detected by our approaches. The second column presents the number of weaknesses of each security design principle. The third, fourth, and fifth columns show the number weaknesses that can be detected by both MCP and MST, only by MCP, and only by MST, respectively. The number of weaknesses that can be discovered by either MCP or MST is shown in the last column. In total, 131 out of 223 (59%) weaknesses from the CWE Security Design Principles can be discovered by our approaches. 101 out of these 131 (77%) weaknesses can be discovered by both MCP and MST. In other words, we can apply either MCP or MST to detect these 101 weaknesses. This mostly depends on the fact that both approaches perform testing by automating user-system interactions. In MCP, user-system interactions are indicated in misuse case specifications in natural language that are manually written by test engineers. Therefore, it is easy to simulate a complex attack scenario that potentially includes a large set of activities performed by a malicious user. Instead, the user-system interactions in MST correspond to the source inputs that are automatically collected by a Web crawler or belong to manually implemented test scripts (e.g., test scripts for functional system testing). Therefore, in the case of MST it is difficult to discover weaknesses that require complex interactions to be tested (e.g., input sequences that include activities from multiple, distinct users). Consequently, MCP can cover all the weaknesses detected by MST (see fifth Column in Table7.23).

We now discuss the weaknesses detected by MCP only, which are 30 out of 131 (23%), as shown in Table 7.23. The four security design principles associated with most of these weaknesses are *Validate Inputs*, *Authorize Actors*, *Audit*, and *Authenticate Actors* (i.e., seven, six, five, and five weaknesses, respectively). The seven weaknesses related to *Validate Inputs* are about injecting harmful code to the system (see Section 7.4). The *Authorize Actors* weaknesses are about retrieving sensitive or private information by means of data analysis (reason R4 in Table 7.16) or about the storage of sensitive data in a mobile operation system (e.g., Android), which is not supported by MST (reason R1 in Table 7.16). Most of the weaknesses related to *Authenticate Actors* concern passwords management (e.g., password aging, empty password); they can be discovered through a single test case execution and thus should not be targeted through MT (reason R5 - MT is inefficient - in Table 7.16). *Audit* weaknesses require data analysis to evaluate the quality of system logs (reason R4, see the discussion in Section 7.4).

Table 7.23. CWE security design principle weaknesses addressed by MCP and MST

Security Design Principle	# weaknesses	# addressed weaknesses			
		both	only MCP	only MST	sum
Audit	6	0	5	0	5
Authenticate Actors	28	15	5	0	20
Authorize Actors	60	32	6	0	38
Cross Cutting	9	5	0	0	5
Encrypt Data	38	5	2	0	7
Identify Actors	12	9	1	0	10
Limit Access	8	3	2	0	5
Limit Exposure	6	0	0	0	0
Lock Computer	1	0	1	0	1
Manage User Sessions	6	4	0	0	4
Validate Inputs	39	26	7	0	33
Verify Message Integrity	10	2	1	0	3
Total	223	101	30	0	131

Table 7.24. OWASP Top 10 security risk weaknesses addressed by MCP and MST

OWASP Security Risk	# weaknesses	# addressed weaknesses			
		both	only MCP	only MST	sum
Injection	9	7	1	0	8
Broken Authentication	9	6	0	0	6
Sensitive Data Exposure	11	4	2	0	6
XML External Entities	2	0	1	0	1
Broken Access Control	5	5	0	0	5
Security Misconfiguration	3	2	0	0	2
Cross-Site Scripting (XSS)	1	0	1	0	1
Insecure Deserialization	1	0	0	0	0
Using Components with Known Vulnerabilities	-	-	-	-	-
Insufficient Logging & Monitoring	2	0	1	0	1
Total	43	24	6	0	30

Table 7.25. CWE Top 25 weaknesses addressed by MCP and MST

# weaknesses	# addressed weaknesses			
	both	only MCP	only MST	sum
25	9	6	0	15

Table 7.24 and Table 7.25 compare MST and MCP based on high-risk weaknesses in OWASP Top 10 security risks and CWE Top 25. MCP can detect all the weaknesses that can be discovered by MST. However, the percentage of weaknesses which can be detected by both MCP and MST is still high (80% for the cases in OWASP Top 10 security risks and 60% for the case of CWE Top 25).

Although MCP is capable of discovering a higher percentage of weaknesses than MST, it presents a major limitation with respect to MST, which is the need for manually specified inputs and oracles. Instead, MST can automatically perform security testing by automatically deriving test inputs from Web-crawler data and do not require manually specified oracles. For this reason, to reduce testing effort, we recommend to apply both MST and MCP.

First, MST should be applied to maximize the number of security vulnerabilities that can be automatically tested with minimal engineering effort. In total, MST enables the identification of 104 weaknesses (see Section 7.4. MCP should then be applied to verify the weaknesses not covered by MST. MCP will enable targeting 33 additional weaknesses (30 belonging to CWE Security Design Principles, 1 to OWASP Top 10 only, 2 to CWE Top 25 only, and 2 belonging to all the three views). In Section 7.7, we discuss to what extent the adoption of both approaches may augment costs related to testability.

7.7 RQ6: Comparison of Testability Guidelines for MCP and MST

7.7.1 Measurements

In this RQ, we aim to investigate if the same testability guidelines enable the application of both MCP and MST. To this end, we identify the testablity features that are required by both MCP and MST for the three CWE views considered in our analysis. Furthermore, we discuss why certain features are required by one approach only. Finally, we discuss which testability features are required we combine the use of MCP and MST as suggested in Section 7.6.

7.7.2 Results

MCP can discover 131 weaknesses in the CWE Security Design Principles view (Table 7.2), with support from 14 testability features (Table 7.10). MST can detect 101 weaknesses in the same CWE view (Table 7.13), with support from 10 testability features (Table 7.20).

In the Security Design Principles view, MCP and MST share 10 testability features. A high percentage (71%, i.e., TF3, TF4, TF5, TF6, TF7, TF8, TF9, TF11, TF13, and TF14) of them enable testing through user-system interactions. Among these 10 testability features, TF3 and TF4 play the most important role. Indeed they enable MCP and MST to discover respectively 88 (67%) or 82 (81%) weaknesses in the CWE view Security Design Principles.

The four testability features which are not required by MST are *TF1*, *TF2*, *TF10*, and *TF12*. We discuss them below.

With TF1, the test framework can access log files in the system thanks to the features provided by the system under test. This testability feature supports MCP to detect five *Audit* weaknesses, two *Encrypt Data* weaknesses, one *Authorize Actors* weakness, and one *Validate Inputs* weakness in the view CWE Security Design Principles (Table 7.10). All these weaknesses can be discovered through the analysis of log files; more precisely, by looking for relations between test inputs and content appearing in the log files. Since these relations cannot be defined in a MR, MST cannot discover these weaknesses and thus TF1 is not needed by MST.

The testability feature TF2 concerns retrieving information from the system under test by means of a regular expression. It supports MCP to detect six weaknesses in the CWE Security Design

Table 7.26. Distribution of testability features of MST and MCP for the security design principles view.

Security Design Principle	Testability Feature Number														Sum
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	
MST															
Audit	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
Authenticate Actors	-	-	3	9	2	-	-	-	1	-	-	-	-	-	15
Authorize Actors	-	-	24	10	-	3	3	1	-	-	-	-	-	-	41
Cross Cutting	-	-	2	2	-	-	-	-	-	-	-	-	-	-	4
Encrypt Data	-	-	1	1	-	-	-	-	3	-	-	-	-	-	5
Identify Actors	-	-	2	2	-	-	-	-	-	-	-	-	-	7	11
Limit Access	-	-	2	-	-	1	-	-	1	-	-	-	1	-	5
Limit Exposure	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
Lock Computer	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
Manage User Sessions	-	-	1	1	-	-	-	-	-	-	1	-	2	-	5
Validate Inputs	-	-	17	15	-	-	-	-	-	-	-	-	-	-	32
Verify Message Integrity	-	-	1	2	-	-	-	-	-	-	-	-	-	-	3
MCP															
Audit	5	2	-	-	-	-	-	-	-	3	-	-	-	-	10
Authenticate Actors	-	-	-	-	2	-	-	-	-	-	2	1	-	-	5
Authorize Actors	1	3	2	-	-	-	-	-	1	-	-	-	-	-	7
Cross Cutting	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
Encrypt Data	-	-	-	-	-	-	-	-	1	-	-	-	-	-	1
Identify Actors	-	-	-	-	-	-	-	-	-	-	-	-	1	-	1
Limit Access	-	1	-	-	-	-	-	-	1	-	-	-	1	-	3
Limit Exposure	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
Lock Computer	-	-	-	-	1	-	-	-	-	-	-	-	-	-	1
Manage User Sessions	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
Validate Inputs	1	-	-	4	-	-	-	-	-	-	-	-	2	-	7
Verify Message Integrity	-	-	1	-	-	-	-	-	-	-	-	-	-	-	1
Total	7	6	56	46	5	4	3	1	8	3	3	1	7	7	157
% of weaknesses *	5%	5%	43%	34%	4%	3%	2%	1%	6%	2%	2%	1%	5%	5%	100%
Total, MST features only	0	0	53	42	2	4	3	1	5	0	1	0	3	7	121
% of weaknesses *	0%	0%	40%	31%	2%	3%	2%	1%	4%	0%	1%	0%	2%	5%	77%

* The percentage of weaknesses that can be discovered thanks to the testability feature.

Principles view (Table 7.10). The regular expression is manually defined by test engineers and is used as an oracle. In contrast, for MST, we do not expect engineers to define regular expressions as oracles since they are derived from the comparison of the output of source and follow-up inputs. Thus, MST does not require TF2.

Three *Audit* weaknesses in the view of CWE Security Design Principles can be discovered by MCP when the testability features TF1 and TF10 are fulfilled. These three weaknesses concern the quality of recorded logs in the system (i.e., insufficient logging or excessive data). However, since MST does not deal with log files (see Paragraph above), MST cannot discover such weaknesses. Therefore, TF10 is not required by MST.

The last testability feature required only by MCP is TF12. It helps MCP to detect one weakness related to the security design principle *Authenticate Actors* (i.e., *use of password hash instead of password for authentication*). To cover this weakness, MCP needs a feature to get stored passwords in the database server. Since such type of testing cannot be expressed in terms of a metamorphic relation (i.e, a password can be replaced by a single hash only), MST does not cover this weakness and, consequently, does not require TF12.

Table 7.27. Distribution of testability features of MST and MCP for the weaknesses associated with the OWASP Top 10 security risks.

OWASP Security Risk	Testability Feature Number														Sum
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
MST															
Injection	-	-	4	5	-	-	-	-	-	-	-	-	-	-	9
Broken Authentication	-	-	2	1	1	-	-	-	1	-	1	-	1	-	7
Sensitive Data Exposure	-	-	1	-	-	-	-	-	2	-	-	-	-	1	4
XML External Entities	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
Broken Access Control	-	-	4	1	-	-	1	-	-	-	-	-	-	-	6
Security Misconfiguration	-	-	2	1	-	-	-	-	-	-	-	-	-	-	3
Cross-Site Scripting	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
Insecure Deserialization	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
Using Components with Known Vulnerabilities	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
Insufficient Logging & Monitoring	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
MCP															
Injection	-	-	1	-	-	-	-	-	-	-	-	-	-	-	1
Broken Authentication	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
Sensitive Data Exposure	-	1	-	-	-	-	-	-	1	-	-	-	-	-	2
XML External Entities	-	-	-	-	-	-	-	-	-	-	-	-	1	-	1
Broken Access Control	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
Security Misconfiguration	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
Cross-Site Scripting	-	-	-	-	-	-	-	-	-	-	-	-	1	-	1
Insecure Deserialization	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
Using Components with Known Vulnerabilities	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
Insufficient Logging & Monitoring	1	-	-	-	-	-	-	-	-	1	-	-	-	-	2
Total	1	1	14	8	1	0	1	0	4	1	1	0	3	1	36
% of weaknesses *	3%	3%	47%	27%	3%	0%	3%	0%	13%	3%	3%	0%	10%	3%	100%
Total, MST features only	0	0	13	8	1	0	1	0	3	0	1	0	1	1	29
% of weaknesses *	0%	0%	43%	27%	3%	0%	3%	0%	10%	0%	3%	0%	3%	3%	80%

* Percentage of weaknesses that can be discovered thanks to a testability feature.

In OWASP Top 10 security risks, MCP and MST share eight common testability features (i.e., TF3, TF4, TF5, TF7, TF9, TF11, TF13 and TF14: see Table 7.11 and Table 7.21). MCP needs three additional testability features (i.e., TF1. TF2, and TF10) to cover six weaknesses not covered by MST. TF3 and TF4 still play the most important role.

According CWE top 25 weaknesses, five testability features (i.e., TF3, TF4, TF7, TF9 and TF14: see Table 7.11 and Table 7.21) are shared between MCP and MST. Two additional testability features (i.e., TF2 and TF13) are required by MCP to discover six weaknesses not detected by MST. TF3 and TF4 remain the most important testability features.

At the end of Section 7.6, we proposed to combine MST and MCP to discover a total of 137 weaknesses, 104 with MST and 33 additional ones with MCP. Tables 7.26 to 7.28 show the testability features required to automate such a process, for each CWE view considered in our analysis. Concerning Security Design Principles view, 10 testability features are required to test 101 weaknesses with MST, and 4 additional features are required by MCP. It can be observed that the 10 testability features required by MST enable the testing of 120 weaknesses (101 with MST, 19 with MCP); thus, MCP can be applied on top of MST without additional costs. Concerning the OWASP Top 10 security risks view, MST needs eight testability features (i.e., TF3, TF4, TF5, TF7, TF9, TF11, TF13, and

Table 7.28. Distribution of testability features of MST and MCP for the CWE Top 25 weaknesses.

CWE Top 25 Weakness	**Testability Feature Number**														**Sum**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	
MST															
Improper Input Validation	-	-	1	1	-	-	-	-	-	-	-	-	-	-	2
SQL Injection	-	-	1	1	-	-	-	-	-	-	-	-	-	-	2
Path Traversal	-	-	1	-	-	-	-	-	-	-	-	-	-	-	1
OS Command Injection	-	-	1	-	-	-	-	-	-	-	-	-	-	-	1
Improper Authentication	-	-	1	-	-	-	-	-	-	-	-	-	-	-	1
Incorrect Permission Assignment for Critical Resource	-	-	1	-	-	-	-	-	-	-	-	-	-	-	1
Unrestricted Upload of File with Dangerous Type	-	-	-	1	-	-	-	-	-	-	-	-	-	-	1
Improper Privilege Management	-	-	1	-	-	-	1	-	-	-	-	-	-	-	2
Improper Certificate Validation	-	-	-	-	-	-	-	-	-	-	-	-	-	1	1
MCP															
Cross-Site Scripting	-	-	-	-	-	-	-	-	-	-	-	-	1	-	1
Information Exposure	-	1	-	-	-	-	-	-	1	-	-	-	-	-	2
Integer Overflow or Wraparound	-	-	-	1	-	-	-	-	-	-	-	-	-	-	1
Cross-Site Request Forgery (CSRF)	-	-	-	-	-	-	-	-	-	-	-	-	1	-	1
Improper Restriction of XML External Entity Reference	-	-	-	-	-	-	-	-	-	-	-	-	1	-	1
Code Injection	-	-	-	1	-	-	-	-	-	-	-	-	-	-	1
Total	0	1	7	5	0	0	1	0	1	0	0	0	3	1	19
% of weaknesses *	0%	7%	47%	33%	0%	0%	7%	0%	7%	0%	0%	0%	20%	7%	100%
Total, MST features only	0	0	7	3	0	0	1	0	0	0	0	0	0	1	12
% of weaknesses *	0%	0%	47%	20%	0%	0%	7%	0%	0%	0%	0%	0%	0%	7%	75%

* The percentage of weaknesses that can be discovered thanks to the testability feature.

TF14) to discover 24 weaknesses, while MCP requires two additional features (i.e., TF1 and TF2) to discover six additional weaknesses. The eight testability features required by MST enable the testing of 28 weaknesses (24 with MST and 4 with MCP). Regarding the CWE Top 25 view, four testability features are required to detect nine weaknesses with MST, and three additional testability features are needed by MCP to discover six more weaknesses. With only four testability features (i.e., TF3, TF4, TF7, and TF14) required by MST, 11 weaknesses can be discovered (nine by MST and two additional by MCP).

Tables 7.26 to 7.28 can be used as testability guidelines for the software engineers who aim to automate testing by combining MST and MCP. Following our suggestion (see Section 7.6), MST should be applied first because it requires minimal inputs from the test engineer. For this reason, implementing the testability features required by MST is highly suggested since MST comes with limited additional costs. Also, we have observed that the testability features required by MST enable the testing of a number of weaknesses discovered by MCP only. Indeed, MCP covers six additional weaknesses belonging to the *Validate Inputs* security design principle with the same set of 10 testability features required by MST. Three of them are high risk weaknesses (i.e., Cross-Site Script, Cross-Site Request Forgery, Code Injection). Though the discovery of such weaknesses requires the definition of complex attack scenarios including interactions between the system under test and multiple users, the RMCM template used by MCP enable the definition of such scenarios. Therefore, MCP complements MST for weaknesses with complex attack scenarios without additional testability costs.

Concerning the additional testability features required by MCP, we highlight that MCP can discover weaknesses related to *Audit*, *Authorize Actors*, *Limit Access*, and *Validate Inputs*. The *Audit* security design principle is not addressed by MST; therefore, if the verification of log quality is important, the adoption of MCP is strongly suggested. The implementation of the additional testability features required by MCP (i.e., TF1, TF2, and TF10) is thus highly justified in this case. If the verification of log file quality is not necessary, engineers may evaluate whether it is more expensive to implement TF1, TF2, and TF10 or manually test the six weaknesses related to *Authorize Actors*, *Limit Access*, and *Validate Inputs*. In general, we suggest to implement TF1, TF2, and TF10 because test automation is an investment that minimizes human errors and pays off on the long term.

Chapter 8

Conclusions and Future Work

In this chapter we summarize the contributions of this dissertation and discuss some perspectives on potential future work in this area.

8.1 Summary

Software systems are pervasive in most areas of society and industry. People continuously interact with software applications (e.g., Apps on a smartphone, browser in a desktop computer) to perform different kinds of important activities (e.g., working, reading, playing games, doing physical exercise). A common characteristic of many of these software systems is that they are Web-oriented, i.e., the computation is performed on a remote system (e.g., a server) and the input data and results are communicated through the Web. With these types of systems, the security and privacy of both software and data are a primary concern for all stakeholders.

Examples of such Web-oriented systems are systems developed in the context of a European project in the healthcare domain named EDLAH2. The objective of the project was to improve the daily activities of elder people through a gamification-based approach. EDLAH2 case systems manage and exchange personal and sensitive information (i.e., health status, personal security information, personal messages and pictures). The work described in this dissertation has been partially motivated by the needs of the EDLAH2 project.

In this dissertation, we focused on the definition of methods and the design of automated solutions to help ensure that *web-oriented software systems* meet their security requirements.

First, we addressed the challenges related to the specification of security requirements in a structured and analyzable form that supports communication among stakeholders, and eases the verification of security requirements. To this end, we defined a requirements specification template and a requirements modelling methodology, called RMCM. RMCM supports the modeling of security requirements in a structured and analyzable manner by means of misuse case diagrams and security use case specifications. A misuse case diagram describes the relations between actors, use cases, misuse cases, and security use cases. To specify security use case specifications, we extended the RUCM modelling method for functional requirements. More precisely, we extended RUCM with rules to specify misuse case specifications and mitigation schemes. Furthermore, we defined a set of NLP-based algorithms to automatically check the consistency between requirement specifications and diagrams, and the conformance between requirement specifications and the proposed template. Empirical results and a questionnaire with the EDLAH2 engineers have shown that RMCM is precise and practical to capture the security requirements of Web-oriented software systems in industrial settings.

Second, we defined solutions to automatically generate executable security test cases from security requirements in natural language. More precisely, our solution processes requirements written according to the RMCM requirements specification template to generate executable test cases that verify the correct implementation of security requirements. Our approach, called *Misuse Case Programming* (MCP), automatically generates executable security test cases from misuse case specifications. More specifically, MCP relies on Natural Language Programming concepts to support test case generation. MCP leverages Natural Language Processing (NLP) techniques to extract the concepts appearing in requirements specifications and generates executable test cases by matching the extracted concepts to the members of a provided test driver API. The evaluation performed with the EDLAH2 case study system provides evidence of the feasibility and benefits of the approach. MCP also reduces the effort required for performing security vulnerability testing since it automates the generation of executable test cases which are not easy to implement manually.

Third, we proposed a solution to contain test automation costs due to the identification of test inputs and associated test oracles. More precisely, we defined a metamorphic security testing approach, MST, to automatically generate test inputs and test oracles for security testing. MST enables engineers to specify metamorphic relations (MR) capturing security properties of web-oriented software systems. To the best of our knowledge, the work described in this dissertation is the first metamorphic testing solution automating the detection of a large set of security vulnerabilities. Our contributions to the state-of-the-art include (1) a DSL and supporting tools for specifying MRs for security testing, (2) a set of MRs inspired by OWASP guidelines, (3) a data collection framework that crawls the system under test to automatically derive input data, and (4) a metamorphic testing framework that automatically detect security vulnerabilities based on the provided MRs. Our analysis of the OWASP guidelines shows that MST can automate 39% of the security testing activities not currently targeted by state-of-the-art techniques. Empirical results performed with three case study systems have shown that the approach requires limited manual effort and detects 85.7% of the targeted vulnerabilities. Moreover, MST allowed us to discover an unknown vulnerability in a widely adopted CI/CD system (Jenkins).

Fourth, we derived testability guidelines to support test engineers in automatically testing software systems with the proposed automated testing approaches (i.e., MCP and MST). More precisely, we investigated which types of security vulnerabilities can be identified by the proposed approaches, and what guidelines should be followed to adopt these approaches as effectively as possible in web-oriented software systems. Our study has shown that MCP and MST address a high percentage of weaknesses reported in the CWE database [CWE, 2020s]. More precisely, they enable the detection of 59% and 45% of the weaknesses related to security design principles, respectively. They can detect 70% and 56% of the OWASP Top 10 security risks, respectively. Also, they can discover 60% and 36% of the CWE Top 25 weaknesses, respectively. In general, MCP can discover a superset of the vulnerabilities detected by MST but entails higher costs due to test input selection and test oracle definition. For this reason, we recommend to apply both MST and MCP; more precisely, we suggest to rely on MCP to discover those vulnerabilities that cannot be detected by MST. Concerning testability, we have identified 14 different features (i.e., testability features) that either the software under test or the test framework should implement to fully leverage the capabilities of MCP and MST. MCP and MST share most of the features, with MCP requiring a larger set. As part of our guidelines, we have identified the subset of testability features required to verify distinct design principles or high-risk weaknesses. Our testability guidelines thus support engineers to determine, based on the security requirements of the system under test, which testability features need to be enabled.

8.2 Future Work

In this dissertation, we focused on the problem of capturing security requirements in a structured and analyzable manner, automating the process of security requirements verification, and addressing the oracle problem in security testing. In the future, we aim to extend and consolidate the techniques developed in the context of this dissertation to make them applicable to a broader set of systems, including cyber-physical, Internet-of-Things (IoT), and telecommunication (e.g., satellite) systems.

Concerning requirements elicitation (i.e., RMCM) and verification (i.e., MCP), future work should focus on introducing more flexibility in the writing of the requirements specifications, which includes the handling of synonyms [Wang et al., 2018], and support for compound sentences. Also, from a more technical standpoint, it is necessary to extend the MCP test driver API to support a broader set of testability features that might be required in different contexts (e.g., IoT and satellite communications).

Concerning metamorphic security testing, we believe that the proposed DSL and toolset can be leveraged by other researchers to extend the set of supported metamorphic relations, including domain specific ones (e.g., to detect vulnerabilities in cyber-physical systems). Also, since the degree of automation of MST largely depends on the capability of the Web crawler used to collect source inputs, future research directions should include the development of Web crawlers aiming at mimicking the behaviour of end-users and capable of exploring a large set of features of the system under test.

List of Papers

Published papers included in this dissertation:

1. Phu X. Mai, Arda Göknil, Lwin Khin Shar, Fabrizio Pastore, Lionel C. Briand, and Shaban Shaame. **“Modeling security and privacy requirements: a use case-driven approach”**. *Information and Software Technology*, 100 (2018): 165-182.

2. Phu X. Mai, Fabrizio Pastore, Arda Göknil, and Lionel C. Briand. **“A natural language programming approach for requirements-based security testing”**. In *Proceedings of the 29th IEEE International Symposium on Software Reliability Engineering (ISSRE 2018)*, Memphis, TN, USA, October 15-18, 2018, pp. 58-69, 2018.

3. Phu X. Mai, Fabrizio Pastore, Arda Göknil, and Lionel C. Briand. **“MCP: A Security Testing Tool Driven by Requirements”**. In *the 41st International Conference on Software Engineering: Companion Proceedings*, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, pp. 55-58, 2019.

4. Phu X. Mai, Fabrizio Pastore, Arda Göknil, and Lionel C. Briand. **“Metamorphic Security Testing for Web Systems”**. In *Proceedings of IEEE International Conference on Software Testing, Verification and Validation (ICST) 2020*, Porto, Portugal, October 24-28, 2020, pp. 186-197, 2020.

5. Phu X. Mai, Arda Göknil, Fabrizio Pastore, and Lionel C. Briand. **“SMRL: A Metamorphic Security Testing Tool for Web Systems”**. In *the 42nd International Conference on Software Engineering: Companion Proceedings*, ICSE 2020, Seoul, Republic of Korea, July 06-11, 2020, pp. 9-12, 2020.

Bibliography

[OWA, 2017a] (2017a). CWE Category: OWASP Top Ten 2017 Category A9 - using components with known vulnerabilities. https://cwe.mitre.org/data/definitions/1035.html.

[Edl, 2017] (2017). EDLAH2 Questionaire. https://goo.gl/forms/OrAZcZLvsVm5tUN13.

[MCP, 2018] (2018). MCP, prototype tool and experimental data. https://sntsvv.github.io/MCP/.

[Web, 2019] (2019). SMRL editor executable, catalog of MRs, MT framework, experimental data. https://sntsvv.github.io/SMRL/.

[Sto, 2020] (2020). CVE-2020-2162: Stored XSS vulnerability in file parameters. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-2162.

[Jen, 2017] (Visited in 2017). Apache Jena Semantic Web and Linked Data toolset. https://jena.apache.org/.

[Dru, 2017] (Visited in 2017). Drupal Content Management System. https://www.drupal.org/.

[Pyt, 2017] (Visited in 2017). Function annotations, https://www.python.org/dev/peps/pep-3107/.

[JUn, 2017] (Visited in 2017). JUnit Testing Framework. https://www.junit.org/.

[MiC, 2017] (Visited in 2017). MiCare. http://www.icare247.eu/edlah2/.

[OWA, 2017b] (Visited in 2017b). Open Web Application Security Project. https://www.owasp.org/.

[SQL, 2017] (Visited in 2017). SQLMap, testing tool focussing on the detection of SQL injections. http://sqlmap.org/.

[GAT, 2017] (Visited in 2017). The GATE workbench. http://gate.ac.uk/.

[OWL, 2017] (Visited in 2017). Web Ontology Language (OWL). https://www.w3.org/OWL/.

[ASM, 2018] (Visited in 2018). ASM bytecode manipulation framework. https://asm.ow2.io/.

[CC, 2018] (Visited in 2018). Common Criteria for Information Technology Security Evaluation. http://www.commoncriteriaportal.org.

[Ecl, 2018] (Visited in 2018). Eclipse IDE, https://www.eclipse.org/ide/.

[OWA, 2018] (Visited in 2018). OWASP, Android Testing Guidelines. https://www.owasp.org/index.php/Android_Testing_Cheat_Sheet.

[web, 2018] (Visited in 2018). Selenium Web Testing Framework, https://www.seleniumhq.org/.

[Xte, 2018] (Visited in 2018). Xtext, https://www.eclipse.org/Xtext/.

[CAP, 2020a] (Visited in 2020a). CAPEC-592: Stored XSS. https://capec.mitre.org/data/definitions/592.html.

[CAP, 2020b] (Visited in 2020b). CAPEC common attack pattern enumeration and classification project, which provide a taxonomy of attacks. https://capec.mitre.org.

[CVS, 2020] (Visited in 2020). CVSS: Common Vulnerability Scoring System. https://www.first.org/cvss/.

[CWE, 2020a] (Visited in 2020a). CWE - Common Weakness Enumeration. https://cwe.mitre.org.

[CWE, 2020b] (Visited in 2020b). CWE-20: Improper input validation. https://cwe.mitre.org/data/definitions/20.html.

[CWE, 2020c] (Visited in 2020c). CWE-223: Omission of security-relevant information. https://cwe.mitre.org/data/definitions/223.html.

[CWE, 2020d] (Visited in 2020d). CWE-224: Obscured security-relevant information by alternate name. https://cwe.mitre.org/data/definitions/224.html.

[CWE, 2020e] (Visited in 2020e). CWE-287: Improper authentication. https://cwe.mitre.org/data/definitions/287.html.

[CWE, 2020f] (Visited in 2020f). CWE-315: Cleartext storage of sensitive information in a cookie. https://cwe.mitre.org/data/definitions/315.html.

[CWE, 2020g] (Visited in 2020g). CWE-613: Insufficient session expiration. https://cwe.mitre.org/data/definitions/613.html.

[CWE, 2020h] (Visited in 2020h). CWE-640: Weak password recovery mechanism for forgotten password. https://cwe.mitre.org/data/definitions/640.html.

[CWE, 2020i] (Visited in 2020i). CWE-79: Improper neutralization of input during web page generation ('cross-site scripting'). https://cwe.mitre.org/data/definitions/79.html.

[CWE, 2020j] (Visited in 2020j). CWE Category: : Sert cei c coding standards. https://cwe.mitre.org/data/definitions/1154.html.

[CWE, 2020k] (Visited in 2020k). CWE Category: : Software fault patterns. https://cwe.mitre.org/data/definitions/888.html.

[CWE, 2020l] (Visited in 2020l). CWE-FAQ. https://cwe.mitre.org/about/faq.html#A.1.

[CWE, 2020m] (Visited in 2020m). CWE Top 25 Most Dangerous Software Errors. https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html.

[CWE, 2020n] (Visited in 2020n). CWE VIEW: Architectural Concepts. https://cwe.mitre.org/data/definitions/1008.html.

[CWE, 2020o] (Visited in 2020o). CWE VIEW: Hardware Design. https://cwe.mitre.org/data/definitions/1194.html.

[CWE, 2020p] (Visited in 2020p). CWE VIEW: Research Concepts. https://cwe.mitre.org/data/definitions/1000.html.

[CWE, 2020q] (Visited in 2020q). CWE VIEW: Software Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[CWE, 2020r] (Visited in 2020r). CWE View: Weaknesses in owasp top ten (2017). https://cwe.mitre.org/data/definitions/1026.html.

[eba, 2020] (Visited in 2020). ebay. https://www.ebay.com.

[Fac, 2020] (Visited in 2020). Facebook. https://www.facebook.com.

[J2E, 2020] (Visited in 2020). Java Platform, Enterprise Edition. https://www.oracle.com/java/technologies/java-ee-glance.html.

[Joo, 2020] (Visited in 2020). Joomla Content Management System (CMS). https://www.joomla.org.

[Lin, 2020] (Visited in 2020). LinkedIn. https://www.linkedin.com.

[CWE, 2020s] (Visited in 2020s). MITRE common weaknesses enumeration project, which provide a taxonomy of vulnerability types. https://cwe.mitre.org.

[Net, 2020] (Visited in 2020). Netflix. https://www.netflix.com.

[OWA, 2020a] (Visited in 2020a). OWASP Top 10 Web Application Security Risks. https://owasp.org/www-project-top-ten/.

[OWA, 2020b] (Visited in 2020b). OWASP Top 10 Web Security Risks with weaknesses descriptions matching the mitre cwe categories. https://cwe.mitre.org/data/definitions/1026.html.

[Spa, 2020] (Visited in 2020). SpaceX. https://www.spacex.com.

[app, 2020] (Visited in 2020). Who is using Jenkins? https://wiki.jenkins.io/pages/viewpage.action?pageId=58001258.

[Alexander, 2002] Alexander, I. (2002). Initial industrial experience of misuse cases in trade-off analysis. In *RE'02*, pages 61–70.

[Alexander, 2003a] Alexander, I. (2003a). Misuse cases help to elicit non-functional requirements. *Computing & Control Engineering Journal*, 14(1):40–45.

[Alexander, 2003b] Alexander, I. (2003b). Misuse cases: Use cases with hostile intent. *IEEE Software*, 20(1):58–66.

[Anthonysamy et al., 2017] Anthonysamy, P., Rashid, A., and Chitchyan, R. (2017). Privacy requirements: present & future. In *2017 IEEE/ACM 39th international conference on software engineering: software engineering in society track (ICSE-SEIS)*, pages 13–22. IEEE.

[Appelt et al., 2013] Appelt, D., Alshahwan, N., and Briand, L. (2013). Assessing the impact of firewalls and database proxies on sql injection testing. In *FITTEST'13*, pages 32–47.

[Appelt et al., 2014] Appelt, D., Nguyen, C. D., Briand, L. C., and Alshahwan, N. (2014). Automated testing for sql injection vulnerabilities: An input mutation approach. In *ISSTA'14*, pages 259–269.

[Arkin et al., 2005] Arkin, B., Stender, S., and McGraw, G. (2005). Software penetration testing. *IEEE Security & Privacy*, 3(1):84–87.

[Armour and Miller, 2001] Armour, F. and Miller, G. (2001). *Advanced Use Case Modeling: Software Systems*. Addison-Wesley.

[Arora et al., 2015a] Arora, C., Sabetzadeh, M., Briand, L. C., and Zimmer, F. (2015a). Automated checking of conformance to requirements templates using natural language processing. *IEEE Transactions on Software Engineering*, 41(10):944–968.

[Arora et al., 2015b] Arora, C., Sabetzadeh, M., Goknil, A., Briand, L. C., and Zimmer, F. (2015b). Change impact analysis for natural language requirements: An nlp approach. In *RE'15*, pages 6–15.

[Arora et al., 2015c] Arora, C., Sabetzadeh, M., Goknil, A., Briand, L. C., and Zimmer, F. (2015c). NARCIA: an automated tool for change impact analysis in natural language requirements. In *ESEC/FSE'15*, pages 962–965.

[Asnar et al., 2007] Asnar, Y., Giorgini, P., Massacci, F., and Zannone, N. (2007). From trust to dependability through risk analysis. In *ARES'07*, pages 19–26.

[Asnar et al., 2011] Asnar, Y., Giorgini, P., and Mylopoulos, J. (2011). Goal-driven risk assessment in requirements engineering. *Requirements Engineering*, 16:101–116.

[Ballard and Biermann, 1979] Ballard, B. W. and Biermann, A. W. (1979). Programming in natural language: "nlc" as a prototype. In *ACM'79*, pages 228–237.

[Barr et al., 2015] Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., and Yoo, S. (2015). The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525.

[Bau et al., 2010] Bau, J., Bursztein, E., Gupta, D., and Mitchell, J. (2010). State of the art: Automated black-box web application vulnerability testing. In *SP'10*, pages 332–345.

[Beckers, 2012] Beckers, K. (2012). Comparing privacy requirements engineering approaches. In *2012 Seventh International Conference on Availability, Reliability and Security*, pages 574–581. IEEE.

[Bekrar et al., 2011] Bekrar, S., Bekrar, C., Groz, R., and Mounier, L. (2011). Finding software vulnerabilities by smart fuzzing. In *ICST'11*, pages 427–430.

[Ben A. Calloni, 2011] Ben A. Calloni, Djenana Campana, N. M. (2011). Embedded Information Systems Technology Support (EISTS). Task Order 0006: Vulnerability Path Analysis and Demonstration (VPAD). Volume 2 - White Box Definitions of Software Fault Patterns. Technical report, LOCKHEED MARTIN INC FORT WORTH TX. `https://apps.dtic.mil/docs/citations/ADB381215`.

[Bertolino et al., 2012] Bertolino, A., Daoudagh, S., Lonetti, F., Marchetti, E., Martinelli, F., and Mori, P. (2012). Testing of PolPA authorization systems. In *AST'12*, pages 8–14.

[Bettini, 2016] Bettini, L. (2016). *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd.

[Blome et al., 2013] Blome, A., Ochoa, M., Li, K., Peroli, M., and Dashti, M. T. (2013). Vera: A flexible model-based vulnerability testing tool. In *ICST'13*, pages 471–478.

[Bortz and Boneh, 2007] Bortz, A. and Boneh, D. (2007). Exposing private information by timing web applications. In *Proceedings of the 16th International Conference on World Wide Web*, pages 621–628.

[Breaux et al., 2014] Breaux, T. D., Hibshi, H., and Rao, A. (2014). Eddy, a formal language for specifying and analyzing data flow specifications for conflicting privacy requirements. *Requirements Engineering*, 19(3):281–307.

[Breaux and Rao, 2013] Breaux, T. D. and Rao, A. (2013). Formal analysis of privacy requirements specifications for multi-tier applications. In *RE'13*, pages 14–23.

[Cailliau and van Lamsweerde, 2013] Cailliau, A. and van Lamsweerde, A. (2013). Assessing requirements-related risks through probabilistic goals and obstacles. *Requirements Engineering*, 18(2):129–146.

[Carvalho et al., 2014] Carvalho, G., Falcão, D., Barros, F., Sampaio, A., Mota, A., Motta, L., and Blackburn, M. (2014). NAT2TESTSCR: Test case generation from natural language requirements based on SCR specifications. *Science of Computer Programming*, 95:275–297.

[Chan et al., 2007a] Chan, W. K., Chen, T. Y., Cheung, S. C., Tse, T., and Zhang, Z. (2007a). Towards the testing of power-aware software applications for wireless sensor networks. In *ADA Europe'07*, pages 84–99.

[Chan et al., 2007b] Chan, W. K., Cheung, S. C., and Leung, K. R. (2007b). A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Services Research*, 4(2):61–81.

[Chen et al., 1998] Chen, T. Y., Cheung, S.-C., and Yiu, S.-M. (1998). Metamorphic testing: a new approach for generating next test cases. Technical report, The Hong Kong University of Science and Technology.

[Chen et al., 2016] Chen, T. Y., Kuo, F., Ma, W., Susilo, W., Towey, D., Voas, J., and Zhou, Z. Q. (2016). Metamorphic testing for cybersecurity. *Computer*, 49(6):48–55.

[Chen et al., 2018] Chen, T. Y., Kuo, F.-C., Liu, H., Poon, P.-L., Towey, D., Tse, T. H., and Zhou, Z. Q. (2018). Metamorphic testing: A review of challanges and opportunities. *ACM Computing Surveys*, 51(1).

[Chen et al., 2009] Chen, T. Y., Kuo, F.-C., Liu, H., and Wang, S. (2009). Conformance testing of network simulators based on metamorphic testing technique. In *FORTE'09*, pages 243–248.

[Cockburn, 2001] Cockburn, A. (2001). *Writing Effective Use Cases*. Addison-Wesley.

[Data2.eu, 2020] Data2.eu (Visited in 2020). GDPR processing index. `https://data2.eu/`.

[de Figueiredo et al., 2006] de Figueiredo, A. L. L., Andrade, W. L., and Machado, P. D. L. (2006). Generating interaction test cases for mobile phone systems from use case specifications. *SIGSOFT Software Engineering Notes*, 31(6):1–10.

[Deahl, 2018] Deahl, D. (2018). Another Facebook Vulnerability. `https://www.theverge.com/2018/11/13/18088904/imperva-facebook-data-vulnerability-user-friends-information-cambridge-ar`

[DeliveryHero, 2017] DeliveryHero (Visited in 2017). System providing food delivery softwrae and services.

[den Braber et al., 2007] den Braber, F., Hogganvik, I., Lund, M. S., Stolen, K., and Vraalsen, F. (2007). Model-based security analysis in seven steps — a guided tour to the CORAS method. *BT Technology Journal*, pages 101–117.

[Deng et al., 2011] Deng, M., Wuyts, K., Scandariato, R., Preneel, B., and Joosen, W. (2011). A privacy threat analysis framework: Supporting the elicitation and fulfillment of privacy requirements. *Requirements Engineering*, 16(1):3–32.

[Deterding et al., 2011] Deterding, S., Dixon, D., Khaled, R., and Nacke, L. (2011). From game design elements to gamefulness: Defining "gamification". In *MindTrek'11*, pages 9–15. ACM.

[Ding et al., 2011] Ding, J., Wu, T., Wu, D., Lu, J. Q., and Hu, X.-H. (2011). Metamorphic testing of a monte carlo modeling program. In *AST'11*, pages 1–7.

[Eclipse Foundation, 2018] Eclipse Foundation (Visited in 2018). Jetty application server. `https://www.eclipse.org/jetty/`.

[Eclipse Foundation, 2020] Eclipse Foundation (Visited in 2020). Jenkins ci/cd server. `https://jenkins.io/`.

[EDLAH2, 2017a] EDLAH2 (2017a). EDLAH2: Active and Assisted Living Programme. `http://www.edlah2.eu/`.

[EDLAH2, 2017b] EDLAH2 (2017b). EDLAH2: Active and Assisted Living Programme. `http://www.aal-europe.eu/projects/edlah2/`.

[Efftinge et al., 2012] Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., and Hanus, M. (2012). Xbase: Implementing domain-specific languages for java. *ACM SIGPLAN Notices - GPCE '12*, 48(3):112–121.

[El-Attar, 2012] El-Attar, M. (2012). Towards developing consistent misuse case models. *Journal of Systems and Software*, 85(2):323–339.

[El-Attar, 2014] El-Attar, M. (2014). Using SMCD to reduce inconsistencies in misuse case models: A subject-based empirical evaluation. *Journal of Systems and Software*, 87:104–118.

[Elahi and Yu, 2007] Elahi, G. and Yu, E. (2007). A goal oriented approach for modeling and analyzing security trade-offs. In *ER'07*, pages 375–390.

[Fabian et al., 2010] Fabian, B., Gurses, S., Heisel, M., Santen, T., and Schmidt, H. (2010). A comparison of security requirements engineering methods. *Requirements Engineering*, 15:7–40.

[Felderer et al., 2011] Felderer, M., Agreiter, B., Zech, P., and Breu, R. (2011). A classification for model-based security testing. In *VALID'11*, pages 109–114.

[Felderer et al., 2016a] Felderer, M., Büchler, M., Johns, M., Brucker, A. D., Breu, R., and Pretschner, A. (2016a). Chapter one - security testing: A survey. volume 101 of *Advances in Computers*, pages 1–51. Elsevier.

[Felderer et al., 2016b] Felderer, M., Zech, P., Breu, R., Büchler, M., and Pretschner, A. (2016b). Model-based security testing: A taxonomy and systematic classification. *Software Testing, Verification and Reliability*, 26(2):119–148.

[Firesmith, 2003] Firesmith, D. G. (2003). Security use cases. *Journal of Object Technology*, 2(3):53–64.

[FitBit, 2017] FitBit (2017). System providing personal training software services.

[Gall, 2008] Gall, N. (2008). WOA: Putting the Web Back in Web Services. `https://blogs.gartner.com/nick_gall/2008/11/19/woa-putting-the-web-back-in-web-services/`.

[Garousi et al., 2019] Garousi, V., Felderer, M., and Kılıçaslan, F. N. (2019). A survey on software testability. *Information and Software Technology*, 108:35–64.

[Giorgini et al., 2005] Giorgini, P., Massacci, F., Mylopoulos, J., and Zannone, N. (2005). Modeling security requirements through ownership, permission and delegation. In *RE'05*, pages 167–176.

[Glaser and Strauss, 1967] Glaser, B. and Strauss, A. (1967). *The Discovery of Grounded Theory*. Aldine Publishing Co.

[Großmann and Seehusen, 2015] Großmann, J. and Seehusen, F. (2015). Combining security risk assessment and security testing based on standards. In *RISK'15*, pages 18–33.

[Gruber, 1993] Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199 – 220.

[Guderlei and Mayer, 2007] Guderlei, R. and Mayer, J. (2007). Towards automatic testing of imaging software by means of random and metamorphic testing. *International Journal of Software Engineering and Knowledge Engineering*, 17(6):757–781.

[Gurses et al., 2006] Gurses, S., Berendt, B., and Santen, T. (2006). Multilateral security requirements analysis for preserving privacy in ubiquitous environments. In *UKDU'06*.

[Gurses and Santen, 2006] Gurses, S. and Santen, T. (2006). Contextualizing security goals—a method for multilateral security requirements elicitation. In *Sicherheit'06*, pages 42–53.

[Guzzoni et al., 2007] Guzzoni, D., Baur, C., and Cheyer, A. (2007). Modeling human-agent interaction with active ontologies. In *Interaction Challenges for Intelligent Assistants, Papers from the 2007 AAAI Spring Symposium*, pages 52–59, Stanford, California, USA. AAAI.

[H. Cunningham et al, 2017] H. Cunningham et al (Visited in 2017). Developing language processing components with gate version 8 (a user guide), `http://gate.ac.uk/sale/tao/tao.pdf`.

[Hafner and Breu, 2008] Hafner, M. and Breu, R. (2008). *Security Engineering for Service-oriented Architectures*. Springer Science & Business Media.

[Hajri et al., 2017a] Hajri, I., Goknil, A., and Briand, L. C. (2017a). A change management approach in product lines for use case-driven development and testing. In *Poster and Tool Track at REFSQ'17*.

[Hajri et al., 2015] Hajri, I., Goknil, A., Briand, L. C., and Stephany, T. (2015). Applying product line use case modeling in an industrial automotive embedded system: Lessons learned and a refined approach. In *MODELS'15*, pages 338–347.

[Hajri et al., 2016] Hajri, I., Goknil, A., Briand, L. C., and Stephany, T. (2016). PUMConf: a tool to configure product specific use case and domain models in a product line. In *FSE'16*, pages 1008–1012.

[Hajri et al., 2017b] Hajri, I., Goknil, A., Briand, L. C., and Stephany, T. (2017b). Incremental reconfiguration of product specific use case models for evolving configuration decisions. In *REFSQ'17*, pages 3–21.

[Hajri et al., 2018a] Hajri, I., Goknil, A., Briand, L. C., and Stephany, T. (2018a). Change impact analysis for evolving configuration decisions in product line use case models. *Journal of Systems and Software*, 139:211–237.

[Hajri et al., 2018b] Hajri, I., Goknil, A., Briand, L. C., and Stephany, T. (2018b). Configuring use case models in product families. *Software and Systems Modeling*, 17(3):939–971.

[Haley et al., 2008] Haley, C., Laney, R., Moffett, J., and Nuseibeh, B. (2008). Security requirements engineering: A framework for representation and analysis. *IEEE Transactions on Software Engineering*, 34(1):133–153.

[Haley et al., 2004] Haley, C. B., Laney, R., Moffett, J. D., and Nuseibeh, B. (2004). Picking battles: the impact of trust assumptions on the elaboration of security requirements. In *iTrust'04*, pages 347–354.

[Haller et al., 2013] Haller, I., Slowinska, A., Neugschwandtner, M., and Bos, H. (2013). Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *USENIX Security'13*, pages 49–64.

[Hatebur et al., 2006] Hatebur, D., Heisel, M., and Schmidt, H. (2006). Security engineering using problem frames. In *ETRICS'06*, pages 238–253.

[Hatebur et al., 2008] Hatebur, D., Heisel, M., and Schmidt, H. (2008). Analysis and component-based realization of security requirements. In *AReS'08*, pages 195–203.

[He et al., 2008] He, K., Feng, Z., and Li, X. (2008). An attack scenario based approach for software security testing at design stage. In *ISCSCT'08*, pages 782–787.

[Hong et al., 2004] Hong, J. I., Ng, J. D., Lederer, S., and Landay, J. A. (2004). Privacy risk models for designing privacy-sensitive ubiquitous computing systems. In *DIS'04*, pages 91–100.

[Huang et al., 2003] Huang, Y.-W., Huang, S.-K., Lin, T.-P., and Tsai, C.-H. (2003). Web application security assessment by fault injection and behavior monitoring. In *WWW'03*, pages 148–159.

[IBM Doors, 2017] IBM Doors (Visited in 2017). IBM Doors. `http://www.ibm.com/software/products/ca/en/ratidoor`.

[Jackson, 2001] Jackson, M. (2001). *Problem Frames: Analysing and Structuring Software Development Problems*. Addison-Wesley.

[Jacobson, 2004] Jacobson, I. (2004). *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

[Jain and Shanbhag, 2012] Jain, A. K. and Shanbhag, D. (2012). Addressing security and privacy risks in mobile applications. *IT Professional*, 14(5):28–33.

[Jiang et al., 2013] Jiang, M., Chen, T. Y., Kuo, F.-C., and Ding, Z. (2013). Testing central processing unit scheduling algorithms using metamorphic testing. In *ICSESS'13*, pages 530–536.

[Johnson, 2003] Johnson, C. (2003). *A Handbook of Accident and Incident Reporting*. Glasgow University Press.

[Jurafsky and Martin, 2017] Jurafsky, D. and Martin, J. H. (2017). *Speech and Language Processing (3rd ed.)*. Prentice Hall, 3 edition.

[Jürjens, 2001] Jürjens, J. (2001). Towards development of secure systems using umlsec. In *FASE'01*, pages 187–200.

[Jürjens, 2002] Jürjens, J. (2002). UMLsec: Extending UML for secure systems development. In *UML'02*, pages 412–425.

[Jürjens, 2003] Jürjens, J. (2003). *Secure Systems Development with UML*. Springer.

[Jürjens, 2005a] Jürjens, J. (2005a). *Secure Systems Development with UML*. Springer Science & Business Media.

[Jürjens, 2005b] Jürjens, J. (2005b). Sound methods and effective tools for model-based security engineering with UML. In *ICSE'05*, pages 322–331.

[Jürjens, 2008a] Jürjens, J. (2008a). Model-based security testing using UMLsec: A case study. *Electronic Notes in Theoretical Computer Science*, 220(1):93–104.

[Jürjens, 2008b] Jürjens, J. (2008b). Model-based security testing using UMLsec: A case study. *Electronic Notes in Theoretical Computer Science*, 220(1):93–104.

[Just and Schweiggert, 2009] Just, R. and Schweiggert, F. (2009). Evaluating testing strategies for imaging software by means of mutation analysis. In *ICSTW'09*, pages 205–209.

[Kalloniatis et al., 2008] Kalloniatis, C., Kavakli, E., and Gritzalis, S. (2008). Addressing privacy requirements in system design: the pris method. *Requirements Engineering*, 13(3):241–255.

[Kals et al., 2006] Kals, S., Kirda, E., Kruegel, C., and Jovanovic, N. (2006). SecuBat: A web vulnerability scanner. In *WWW'06*, pages 247–246.

[Kaplan et al., 2008] Kaplan, M., Klinger, T., Paradkar, A. M., Sinha, A., Williams, C., and Yilmaz, C. (2008). Less is more: A minimalistic approach to UML model-based conformance test generation. In *ICST'08*, pages 82–91.

[Khamaiseh and Xu, 2017] Khamaiseh, S. and Xu, D. (2017). Software security testing via misuse case modeling. In *DASC/PiCom/DataCom/CyberSciTech'17*, pages 534–541.

[Kim and Cha, 2012] Kim, Y.-G. and Cha, S. (2012). Threat scenario-based security risk analysis using use case modeling in information systems. *Security and Communication Networks*, 5(3):293–300.

[Kohnfelder and Garg, 1999] Kohnfelder, L. and Garg, P. (1999). The STRIDE Threat Model. `https://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx`.

[Kulak and Guiney, 2003] Kulak, D. and Guiney, E. (2003). *Use Cases: Requirements in Context*. Addison-Wesley.

[Kuo et al., 2011a] Kuo, F.-C., Chen, T. Y., and Tam, W. K. (2011a). Testing embedded software by metamorphic testing: A wireless metering system case study. In *LCN'11*, pages 291–294.

[Kuo et al., 2011b] Kuo, F.-C., Liu, S., and Chen, T. Y. (2011b). Testing a binary space partitioning algorithm with metamorphic testing. In *SAC'11*, pages 1482–1489.

[Landhausser et al., 2017] Landhausser, M., Weigelt, S., and Tichy, W. F. (2017). NLCI: a natural language command interpreter. *Automated Software Engineering*, 24(4):839–861.

[Lane, 2003] Lane, D. (2003). Online statistics education: A multimedia course of study.

[Larman, 2002] Larman, C. (2002). *Applying UML and Patterns:An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall Professional.

[Le et al., 2014] Le, V., Afshari, M., and Su, Z. (2014). Compiler validation via equivalence modulo inputs. *ACM SIGPLAN Notices*, 49(6):216–226.

[Le et al., 2013] Le, V., Gulwani, S., and Su, Z. (2013). SmartSynth: Synthesizing smartphone automation scripts from natural language. In *MobiSys'13*, pages 193–206.

[Le Traon et al., 2007] Le Traon, Y., Mouelhi, T., and Baudry, B. (2007). Testing security policies: Going beyond functional testing. In *ISSRE'07*, pages 93–102.

[Lebeau et al., 2013] Lebeau, F., Legeard, B., Peureux, F., and Vernotte, A. (2013). Model-based vulnerability testing for web applications. In *ICSTW'13*, pages 445–452.

[Levenshtein, 1966] Levenshtein, V. I. (1966). Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10.

[Lin et al., 2004] Lin, L., Nuseibeh, B., Ince, D., and Jackson, M. (2004). Using abuse frames to bound the scope of security problems. In *RE'04*, pages 354–355.

[Lin et al., 2003] Lin, L., Nuseibeh, B., Ince, D., Jackson, M., and Moffett, J. (2003). Introducing abuse frames for analysing security requirements. In *RE'03*, pages 371–372.

[Liu et al., 2014] Liu, H., Kuo, F.-C., Towey, D., and Chen, T. Y. (2014). How effectively does metamorphic testing alleviate the oracle problem? *IEEE Transactions on Software Engineering*, 40(1):4–22.

[Liu et al., 2003] Liu, L., Yu, E., and Mylopoulos, J. (2003). Security and privacy requirements analysis within a social setting. In *RE'03*, pages 151–161.

[Liverani, 2018] Liverani, R. S. (Visited in 2018). Integration of burp suite and crawljax. `https://github.com/portswigger/burp-csj`.

[Lodderstedt et al., 2002] Lodderstedt, T., Basin, D. A., and Doser, J. (2002). SecureUML: A UML-based modeling language for model-driven security. In *UML'02*, pages 426–441.

[Maalej et al., 2018] Maalej, W., Amyot, D., and Ruhe, G. (2018). Welcome message from the RE18 chairs. In *2018 IEEE 26th International Requirements Engineering Conference (RE)*, pages 13–16. IEEE.

[Mai et al., 2020a] Mai, P. X., Goknil, A., Pastore, F., and Briand, L. C. (2020a). SMRL: A metamorphic security testing tool for web systems. In *ICSE'20*, pages 9–12.

[Mai et al., 2018a] Mai, P. X., Goknil, A., Shar, L. K., Pastore, F., Briand, L. C., and Shaame, S. (2018a). Modeling security and privacy requirements: a use case-driven approach. *Information and Software Technology*, 100:165–182.

[Mai et al., 2020b] Mai, P. X., Pastore, F., Goknil, A., and Briand, L. (2020b). Metamorphic security testing for web systems. In *IEEE International Conference on Software Testing, Verification and Validation (ICST) 2020*, pages 186–197. IEEE.

[Mai et al., 2018b] Mai, P. X., Pastore, F., Goknil, A., and Briand, L. C. (2018b). A natural language programming approach for requirements-based security testing. In *ISSRE'18*, pages 58–69.

[Mai et al., 2019] Mai, P. X., Pastore, F., Goknil, A., and Briand, L. C. (2019). MCP: a security testing tool driven by requirements. In *ICSE'19*, pages 55–58.

[Mai, 2017] Mai, X. P. (2017). RMCM-V: a tool for checking consistencies between misuse case diagram, specifications, and restricted misuse case modeling templates. `https://sites.google.com/site/rmcmverifier/`.

[Manning et al., 2014] Manning, C., Surdeanu, M., Bauer, J. abd Finkel, J., Bethard, S., and McClosky, D. (2014). The stanford CoreNLP natural language processing toolkit. In *ACL'14*, pages 55–60.

[Marback et al., 2013] Marback, A., Do, H., He, K., Kondamarri, S., and Xu, D. (2013). A threat model-based approach to security testing. *Software: Practice and Experience*, 43(2):241–258.

[Martin and Xie, 2007a] Martin, E. and Xie, T. (2007a). Automated test generation for access control policies via change-impact analysis. In *SESS'07*.

[Martin and Xie, 2007b] Martin, E. and Xie, T. (2007b). A fault model and mutation testing of access control policies. In *WWW'07*, pages 667–676.

[Martin et al., 2006] Martin, E., Xie, T., and Yu, T. (2006). Defining and measuring policy coverage in testing access control policies. In *ICICS'06*, pages 139–158.

[Martin and Lam, 2008] Martin, M. and Lam, M. S. (2008). Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In *USENIX Security'08*, pages 31–43.

[Masood et al., 2010a] Masood, M., Ghafoor, A., and Mathur, A. (2010a). Conformance testing of temporal role-based access control systems. *IEEE Transactions on Dependable and Secure Computing*, 7(2):144–158.

[Masood et al., 2010b] Masood, M., Ghafoor, A., and Mathur, A. (2010b). Conformance testing of temporal role-based access control systems. *IEEE Transactions on Dependable and Secure Computing*, 7(2):144–158.

[Mayer and Guderlei, 2006] Mayer, J. and Guderlei, R. (2006). On random testing of image processing applications. In *QSIC'06*, pages 85–92.

[Mayer et al., 2007] Mayer, N., Dubois, E., and Rifaut, A. (2007). Requirements engineering for improving business/it alignment in security risk management methods. In *Enterprise Interoperability II*, pages 15–26.

[McDermott, 2001] McDermott, J. (2001). Abuse-case-based assurance arguments. In *ACSAC'01*.

[McDermott and Fox, 1999] McDermott, J. and Fox, C. (1999). Using abuse case models for security requirements analysis. In *ACSAC'99*.

[Mead et al., 2005] Mead, N. R., Hough, E. D., and Stehney, T. R. (2005). Security quality requirements engineering (square) methodology. Cmu/sei-2005-tr-009, Carnegie Mellon Software Engineering Institute.

[Mellado et al., 2010] Mellado, D., Blanco, C., Sanchez, L. E., and Fernandez-Medina, E. (2010). A systematic review of security requirements engineering. *Computer Standards & Interfaces*, 32:153–165.

[Mellado et al., 2006a] Mellado, D., Fernandez-Medina, E., and Piattini, M. (2006a). Applying a security requirements engineering process. In *ESORICS'06*, pages 192–206.

[Mellado et al., 2006b] Mellado, D., Fernandez-Medina, E., and Piattini, M. (2006b). A comparison of the Common Criteria with proposals of information systems security requirements. In *ARES'06*, pages 654–661.

[Mesbah et al., 2008] Mesbah, A., Bozdag, E., and Van Deursen, A. (2008). Crawling ajax by inferring user interface state changes. In *ICWE'08*, pages 122–134.

[Mesbah et al., 2012] Mesbah, A., Van Deursen, A., and Lenselink, S. (2012). Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web*, 6(1):3.

[Meucci and Muller, 2014] Meucci, M. and Muller, A. (2014). OWASP Testing Guide v4. `https://www.owasp.org/images/1/19/OTGv4.pdf`.

[Microsoft Corp.,] Microsoft Corp. Silverlight plug-ins and development tools. `https://www.microsoft.com/silverlight/`.

[MITRE, 2018a] MITRE (Visited in 2018a). CVE-2018-1000406, concerns OTG-AUTHN-001. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1000406`.

[MITRE, 2018b] MITRE (Visited in 2018b). CVE-2018-1000409, concerns OTG-SESS-003. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1000409`.

[MITRE, 2018c] MITRE (Visited in 2018c). CVE-2018-1999003, concerns OTG-AUTHZ-002. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999003`.

[MITRE, 2018d] MITRE (Visited in 2018d). CVE-2018-1999004, concerns OTG-AUTHZ-002. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999004`.

[MITRE, 2018e] MITRE (Visited in 2018e). CVE-2018-1999006, concerns OTG-AUTHZ-002. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999006`.

[MITRE, 2018f] MITRE (Visited in 2018f). CVE-2018-1999045, concerns OTG-AUTHZ-002. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999045`.

[MITRE, 2018g] MITRE (Visited in 2018g). CVE-2018-1999046, concerns OTG-AUTHZ-002. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999046`.

[MITRE, 2018h] MITRE (Visited in 2018h). CVE-2018-1999047, concerns OTG-AUTHZ-002. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999047`.

[MITRE, 2020a] MITRE (Visited in 2020a). CVE-2018-11327. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-11327`.

[MITRE, 2020b] MITRE (Visited in 2020b). CVE-2018-17857. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-17857`.

[MITRE Corporation, 2018] MITRE Corporation (Visited in 2018). MITRE Corporation. `https://www.mitre.org`.

[MITRE Corporation, 2020] MITRE Corporation (Visited in 2020). Common vulnerabilities and exposures. `https://cve.mitre.org/cve/`.

[Mouelhi et al., 2008] Mouelhi, T., Fleurey, F., Baudry, B., and Le Traon, Y. (2008). A model-based framework for security policy specification, deployment and testing. In *MODELS'08*, pages 537–552.

[Mouelhi et al., 2009] Mouelhi, T., Le Traon, Y., and Baudry, B. (2009). Transforming and selecting functional test cases for security policy testing. In *ICST'09*, pages 171–180.

[Mouratidis and Giorgini, 2007] Mouratidis, H. and Giorgini, P. (2007). Secure tropos: a security-oriented extension of the tropos methodology. *International Journal of Software Engineering and Knowledge Engineering*, 17(2):285–309.

[Murphy et al., 2008] Murphy, C., Kaiser, G. E., and Hu, L. (2008). Properties of machine learning applications for use in metamorphic testing. Technical report, Columbia University.

[Murphy et al., 2011] Murphy, C., Raunak, M. S., King, A., Chen, S., Imbriano, C., Kaiser, G., Lee, I., Sokolsky, O., Clarke, L., and Osterweil, L. (2011). On effective testing of health care simulation software. In *SEHC'11*, pages 40–47.

[Murphy et al., 2009] Murphy, C., Shen, K., and Kaiser, G. (2009). Using jml runtime assertion checking to automate metamorphic testing in applications without test oracles. In *ICST'09*, pages 436–445.

[Needleman and Wunsch, 1970] Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453.

[Nintendo, 2020] Nintendo (Visited in 2020). Nintendo. `https://www.nintendo.se`.

[Ognawala et al., 2016] Ognawala, S., Ochoa, M., Pretschner, A., and Limmer, T. (2016). MACKE: Compositional analysis of low-level vulnerabilities with symbolic execution. In *ASE'16*, pages 780–785.

[Omoronyia et al., 2011] Omoronyia, I., Salehie, M., Ali, R., Kaiya, H., and Nuseibeh, B. (2011). Misuse case techniques for mobile privacy. In *PriMo'11*.

[Opdahl and Sindre, 2009] Opdahl, A. L. and Sindre, G. (2009). Experimental comparison of attack trees and misuse cases for security threat identification. *Information and Software Technology*, 51:916–932.

[Oppenheim, 2005] Oppenheim, A. N. (2005). *Questionnaire Design, Interviewing and Attitude Measurement*. Continuum.

[OWASP, 2016] OWASP (2016). OWASP Top 10 Mobile Security Risks. https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10.

[OWASP, 2017a] OWASP (Visited in 2017a). OTG-AUTHN-002: Testing for default credentials. https://www.owasp.org/index.php/Testing_for_default_credentials_(OTG-AUTHN-002).

[OWASP, 2017b] OWASP (Visited in 2017b). OTG-BUSLOGIC-006: Testing for the circumvention of workflows. https://www.owasp.org/index.php/Testing_for_the_Circumven\protect\discretionary{\char\hyphenchar\font}{}{}tion_of_Work_Flows_(OTG-BUSLOGIC-006).

[OWASP, 2017c] OWASP (Visited in 2017c). OTG-INFO-010: Mapping application architecture. https://www.owasp.org/index.php/Map_Application_Architecture_(OTG-INFO-010).

[OWASP, 2017d] OWASP (Visited in 2017d). OTG-INPVAL-014: Testing for Buffer Overflow. https://www.owasp.org/index.php/Testing_for_Buffer_Overflow_(OTG-INPVAL-014).

[Palmer et al., 2005] Palmer, M., Gildea, D., and Kingsbury, P. (2005). The proposition bank: An annotated corpus of semantic roles. *Computational Linguistics*, 31(1):71–106.

[Papyrus, 2017] Papyrus (Visited in 2017). Papyrus. https://www.eclipse.org/papyrus.

[Pasquale et al., 2016] Pasquale, L., Spoletini, P., Salehie, M., Cavallaro, L., and Nuseibeh, B. (2016). Automating trade-off analysis of security requirements. *Requirements Engineering*, 21(4):481–504.

[Pezze and Zhang, 2014] Pezze, M. and Zhang, C. (2014). Automated test oracles: A survey. *Advances in Computers*, 95:1–48.

[Portswigger, 2018a] Portswigger (Visited in 2018a). Burp suite. https://portswigger.net/burp.

[Portswigger, 2018b] Portswigger (Visited in 2018b). Burp suite scanning (crawling) feature. https://portswigger.net/burp/documentation/desktop/scanning.

[Portswigger, 2018c] Portswigger (Visited in 2018c). Using burp suite to test for bypass authorization schema using site maps. https://support.portswigger.net/customer/portal/articles/1969842-using-burp-s-%22request-in-browser%22-function-to-test-for-access-control-issues.

[Potter and McGraw, 2004] Potter, B. and McGraw, G. (2004). Software security testing. *IEEE Security & Privacy*, 2(5):81–85.

[Pulido-Prieto and Juárez-Martínez, 2017] Pulido-Prieto, O. and Juárez-Martínez, U. (2017). A survey of naturalistic programming technologies. *ACM Computing Surveys*, 50(5):70:1–70:35.

[Raghavan and Garcia-Molina, 2000] Raghavan, S. and Garcia-Molina, H. (2000). Crawling the hidden web. In *VLDB'01*, pages 129–138.

[Rannenberg et al., 1999] Rannenberg, K., Pfitzmann, A., and Müller, G. (1999). IT security and multilateral security. *Multilateral Security in Communications–Technology, Infrastructure, Economy*, pages 21–29.

[Rashid et al., 2016] Rashid, A., Naqvi, S. A. A., Ramdhany, R., Edwards, M., Chitchyan, R., and Babar, M. A. (2016). Discovering unkown known security requirements. In *ICSE'16*, pages 866–876.

[Rosado et al., 2009] Rosado, D. G., Fernandez-Medina, E., and Lopez, J. (2009). Applying a UML extension to build use cases diagrams in a secure mobile grid application. In *ER'09 Workshops*, pages 126–136.

[Rosen, 2018] Rosen, G. (2018). Facebook Security Update on 'View As' Vulnerability. `https://newsroom.fb.com/news/2018/09/security-update/`.

[Rostad, 2006] Rostad, L. (2006). An extended misuse case notation: Including vulnerabilities and the insider threat. In *REFSQ'06*, pages 33–43.

[Salas and Martins, 2014] Salas, M. and Martins, E. (2014). Security testing methodology for vulnerabilities detection of XSS in web services and WS-security. *ENTCS*, pages 133–154.

[Salini and Kanmani, 2012] Salini, P. and Kanmani, S. (2012). Survey and analysis on security requirements engineering. *Computers and Electrical Engineering*, 38:1785–1797.

[Santos et al., 2017a] Santos, J. C., Peruma, A., Mirakhorli, M., Galstery, M., Vidal, J. V., and Sejfia, A. (2017a). Understanding software vulnerabilities related to architectural security tactics: An empirical investigation of chromium, php and thunderbird. In *ICSA'17*, pages 69–78.

[Santos et al., 2017b] Santos, J. C., Tarrit, K., and Mirakhorli, M. (2017b). A catalog of security architecture weaknesses. In *ICSAW'17*, pages 220–223.

[Schneier, 1999] Schneier, B. (1999). Modeling security threats. *Dr. Dobb's journal*, 24(12).

[Segura et al., 2016] Segura, S., Fraser, G., Sanchez, A. B., and Ruiz-Cortes, A. (2016). A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824.

[Silva et al., 2008] Silva, J. L., Campos, J. C., and Paiva, A. C. R. (2008). Model-based user interface testing with spec explorer and concurtasktrees. *Electronic Notes in Theoretical Computer Science*, 208:77–93.

[Sindre, 2007] Sindre, G. (2007). Mal-activity diagrams for capturing attacks on business processes. In *REFSQ'07*, pages 355–366.

[Sindre et al., 2003] Sindre, G., Firesmith, D. G., and Opdahl, A. L. (2003). A reuse-based approach to determining security requirements. In *REFSQ'03*.

[Sindre and Opdahl, 2001] Sindre, G. and Opdahl, A. L. (2001). Templates for misuse case description. In *REFSQ'01*.

[Sindre and Opdahl, 2005] Sindre, G. and Opdahl, A. L. (2005). Eliciting security requirements with misuse cases. *Requirements Engineering*, 10:34–44.

[Sindre et al., 2002] Sindre, G., Opdahl, A. L., and Brevik, G. F. (2002). Generalization/specialization as a structuring mechanism for misuse cases. In *SREIS'02*.

[Song et al., 2005] Song, E., Reddy, R., France, R., Ray, I., Georg, G., and Alexander, R. (2005). Verifiable composition of access control and application features. In *SACMAT'05*, page 120–129.

[Souag et al., 2016] Souag, A., Mazo, R., Salinesi, C., and Comny-Wattiau, I. (2016). Reusable knowledge in security requirements engineering: a systematic mapping study. *Requirements Engineering*, 21:251–283.

[Spiekermann and Cranor, 2008] Spiekermann, S. and Cranor, L. F. (2008). Engineering privacy. *IEEE Transactions on software engineering*, 35(1):67–82.

[Spotify, 2017] Spotify (Visited in 2017). System providing music streaming software and services.

[Staats et al., 2011] Staats, M., Whalen, M. W., and Heimdahl, M. P. (2011). Programs, tests, and oracles: the foundations of testing revisited. In *ICSE'11*, pages 391–400.

[Sun et al., 2011] Sun, C.-a., Wang, G., Mu, B., Liu, H., Wang, Z., and Chen, T. Y. (2011). Metamorphic testing for web services: Framework and a case study. In *ICWS'11*, pages 283–290.

[Sunwave, 2020] Sunwave (Visited in 2020). System providing healthcare servivices.

[Swiderski, 2004] Swiderski, F. (2004). *Threat modeling*. Microsoft Press.

[Synopsys Inc., 2018] Synopsys Inc. (Visited in 2018). Description of the openssl heartbleed vulnerability. `http://heartbleed.com/`.

[Takanen et al., 2018] Takanen, A., Demott, J. D., Miller, C., and Kettunen, A. (2018). *Fuzzing for Software Security Testing and Quality Assurance*. Artech House.

[Tao et al., 2010] Tao, Q., Wu, W., Zhao, C., and Shen, W. (2010). An automatic testing approach for compiler based on metamorphic testing technique. In *APSEC'10*, pages 270–279.

[The PHP Group, 2017] The PHP Group (Visited in 2017). Php programming language. `http://php.net/`.

[Thomas et al., 2014] Thomas, K., Bandara, A. K., Price, B. A., and Nuseibeh, B. (2014). Distilling privacy requirements for mobile applications. In *ICSE'14*, page 871–882.

[Thummalapenta et al., 2012] Thummalapenta, S., Sinha, S., Singhania, N., and Chandra, S. (2012). Automating test automation. In *ICSE'12*, pages 881–891.

[Tian-yang et al., 2010] Tian-yang, G., Yin-Sheng, S., and You-yuan, F. (2010). Research on software security testing. *World Academy of Science, Engineering and Technology*, 70:647–651.

[Tondel et al., 2008] Tondel, I. A., Jaatun, M. G., and Meland, P. H. (2008). Security requirements for the rest of us: A survey. *IEEE Software*, 25(1):20–27.

[Tripp et al., 2013] Tripp, O., Weisman, O., and Guy, L. (2013). Finding your way in the testing jungle: A learning approach to web security testing. In *ISSTA'13*, pages 347–357.

[Tse and Yau, 2004] Tse, T. and Yau, S. S. (2004). Testing context-sensitive middleware-based software applications. In *COMPSAC'04*, pages 458–466.

[Turpe, 2017] Turpe, S. (2017). The trouble with security requirements. In *RE'17*, pages 122–133.

[University of Illinois, 2017] University of Illinois (2017). CogComp NLP Pipeline.

[USC Credit Union, 2017] USC Credit Union (Visited in 2017). System providing home-banking.

[van Lamsweerde, 2004] van Lamsweerde, A. (2004). Elaborating security requirements by construction of intentional anti-models. In *ICSE'04*, pages 148–157.

[van Lamsweerde, 2009] van Lamsweerde, A. (2009). *Requirements Engineering: from System Goals to UML Models to Software Specifications*. John Wiley and Sons.

[Veanes et al., 2005] Veanes, M., Campbell, C., Schulte, W., and Tillmann, N. (2005). Online testing with model programs. In *ESEC/FSE'13*, pages 273–282.

[Voas and Miller, 1995] Voas, J. M. and Miller, K. W. (1995). Software testability: The new verification. *IEEE software*, 12(3):17–28.

[Wang et al., 2017] Wang, C., Pastore, F., and Briand, L. (2017). System testing of timing requirements based on use cases and timed automata. In *ICST'17*, pages 299–309. IEEE.

[Wang et al., 2018] Wang, C., Pastore, F., and Briand, L. (2018). Automated generation of constraints from use case specifications to support system testing. In *ICST'18*.

[Wang et al., 2020] Wang, C., Pastore, F., Goknil, A., and Briand, L. (2020). Automatic generation of acceptance test cases from use case specifications: an nlp-based approach. *IEEE Transactions on Software Engineering*.

[Wang et al., 2015a] Wang, C., Pastore, F., Goknil, A., Briand, L. C., and Iqbal, M. Z. Z. (2015a). Automatic generation of system test cases from use case specifications. In *ISSTA'15*, pages 385–396.

[Wang et al., 2015b] Wang, C., Pastore, F., Goknil, A., Briand, L. C., and Iqbal, M. Z. Z. (2015b). UMTG: a toolset to automatically generate system test cases from use case specifications. In *ESEC/FSE'15*, pages 942–945.

[Whittle et al., 2008] Whittle, J., Wijesekera, D., and Hartong, M. (2008). Executable misuse cases for modeling security concerns. In *ICSE'08*, pages 121–130.

[Wimmel and Jürjens, 2002] Wimmel, G. and Jürjens, J. (2002). Specification-based test generation for security-critical systems using mutations. In *ICFEM'02*, pages 471–482.

[Xie et al., 2009] Xie, X., Ho, J., Murphy, C., Kaiser, G., Xu, B., and Chen, T. Y. (2009). Application of metamorphic testing to supervised classifiers. In *QSIC'09*, pages 135–144.

[Xu and Nygard, 2006] Xu, D. and Nygard, K. E. (2006). Threat-driven modeling and verification of secure software using aspect-oriented petri nets. *IEEE Transactions on Software Engineering*, 32(4):265–278.

[Xu et al., 2012a] Xu, D., Thomas, L., Kent, M., Mouelhi, T., and Le Traon, Y. (2012a). A model-based approach to automated testing of access control policies. In *SACMAT'12*, pages 209–218.

[Xu et al., 2012b] Xu, D., Tu, M., Sanford, M., Thomas, L., Woodraska, D., and Xu, W. (2012b). Automated security test generation with formal threat models. *IEEE Transactions on Dependable and Secure Computing*, 9(4):526–540.

[Xu et al., 2015] Xu, D., Xu, W., Kent, M., Thomas, L., and Wang, L. (2015). An automated test generation technique for software quality assurance. *IEEE Transactions on Reliability*, 64(1):247–268.

[Yue et al., 2013] Yue, T., Briand, L. C., and Labiche, Y. (2013). Facilitating the transition from use case models to analysis models: Approach and experiments. *ACM Transactions on Software Engineering and Methodology*, 22(1):1–38.

[Zeller et al., 2019] Zeller, A., Gopinath, R., Böhme, M., Fraser, G., and Holler, C. (2019). The fuzzing book. In *The Fuzzing Book*. Saarland University. Retrieved 2019-09-09 16:42:54+02:00.

[Zhou et al., 2012] Zhou, Z. Q., Zhang, S., Hagenbuchner, M., Tse, T., Kuo, F.-C., and Chen, T. Y. (2012). Automated functional testing of online search services. *Software: Testing, Verification and Reliability*, 22(4):221–243.

[Zhu, 2015] Zhu, H. (2015). Jfuzz: A tool for automated java unit testing based on data mutation and metamorphic testing methods. In *TSA'15*, pages 8–15.

Appendix A

Misuse Case Specifications

In this chapter, we list 13 misuse case specifications that follow the RMCM template. Twelve of them were derived from the OWASP testing guidelines [Meucci and Muller, 2014]; an additional specification is dedicated to the EDLAH2 system.

These 13 misuse case specifications were used as input for MCP to test the EDLAH2 system (see Section 5.8). Besides, we present two Mitigation Schemes (in Section A.3) which are used to mitigate these misuse cases.

A.1 Misuse Case Specifications Derived from OWASP Testing Guidelines

1. Access Directory Traversal

MISUSE CASE Access directory traversal
Description The misuse case simulates the OWASP testing activity OTG-AUTHZ-001 (i.e., Testing Directory traversal/file include) [Meucci and Muller, 2014]. The MALICIOUS user tries to access a system file by passing the file path in a URL parameter. File paths are provided by the malicious user in a text file.
Precondition There exists a file containing the list of file paths being requested.
Primary Actor MALICIOUS user
Secondary Actors None
Dependency None
Generalization None
Threats Log In, Query Client Info, Create Account, Query Statistical Analysis Report
Assets system DATA
Basic Threat Flow

```
1. FOREACH url
2. DO
3. The MALICIOUS user SENDS FUZZ VALUES TO the system THROUGH the URL.
4. The system SENDS a response page TO the MALICIOUS user.
5. The MALICIOUS user EXPLOITS the system USING the url.
6. UNTIL the MALICIOUS user used all fuzz values.
7. ENDFOR.
```

Postcondition The MALICIOUS user may have accessed sensitive files or folders on the Web server.
Specific Alternative Flow SAF1

```
RFS 5
1. IF the response page contains the error message THEN
2. RESUME STEP 6.
3. ENDIF.
```

Postcondition The MALICIOUS user cannot access sensitive files or folders on the Web server.
Mitigation Scheme Secure Coding for Web App

2. **Bypass Authentication Schema**

MISUSE CASE Bypass Authentication Schema
Description The misuse case simulates the OWASP testing activity OTG-AUTHN-004 (i.e., Testing for bypassing authentication schema) [Meucci and Muller, 2014]. Precisely, this misuse case replicates the direct page request method (i.e., attempt to directly access a protected page).
Precondition None
Primary Actor MALICIOUS user
Secondary Actors None
Dependency None
Generalization None
Threats Query Client Info, Create Account, Query Statistical Analysis Report
Assets client DATA
Basic Threat Flow

```
1. FOREACH resource
2. The MALICIOUS user REQUESTS the resource FROM the system
3. The system SENDS the response page TO the malicious user
4. The MALICIOUS user EXPLOITS the system USING the resource
5. ENDFOR
```

Postcondition The MALICIOUS user has executed a function dedicated to an authenticated user
Specific Alternative Threat Flow SATF1

```
RFS 2
1. IF the resource contains the role parameter in the URL THEN
2. The MALICIOUS user MODIFIES the role values IN the URL
3. RESUME STEP 2
4. ENDIF
```

Postcondition The MALICIOUS user has executed a function dedicated to an authenticated user
Specific Alternative Threat Flow SATF2

```
RFS 5
1. IF the resource contains the role parameter in the HTTP post data THEN
2. The MALICIOUS user MODIFIES the role values IN the HTTP post data THEN
3. RESUME STEP 2
4. ENDIF
```

Postcondition The MALICIOUS user has executed a function dedicated to an authenticated user
Specific Alternative Flow SAF1

```
RFS 4
1. IF the response page contains the error message THEN
2. ABORT
3. ENDIF
```

Postcondition The MALICIOUS user CANNOT execute a function dedicated to an authenticated user
Mitigation Scheme Secure Coding for Web App

3. **Bypass Authorization Schema**

MISUSE CASE Bypass Authorization Schema
Description The misuse case simulates the OWASP testing activity OTG-AUTHZ-002 (i.e., Testing for bypassing authorization schema) [Meucci and Muller, 2014]. Precisely, this misuse case attempts to access to resources dedicated to a different role.
Precondition At least two client accounts have already been created in the system.
Primary Actor MALICIOUS user
Secondary Actors None
Dependency None
Generalization None
Threats Query Client Info, Create Account, Query Statistical Analysis Report
Assets client DATA
Basic Threat Flow

```
1. FOREACH role
2. The MALICIOUS user SENDS the username and the password TO the system THROUGH the Login page
3. FOREACH resource
4. The MALICIOUS user REQUESTS the resource FROM the system
5. The system SENDS the response page TO the malicious user
6. The MALICIOUS user EXPLOITS the system USING the role and the resource
7. ENDFOR
8. ENDFOR
```

Postcondition The MALICIOUS user has executed a function dedicated to another user with different role
Specific Alternative Threat Flow SATF1
RFS 4

```
1. IF the resource contains the role parameter in the URL THEN
2. The MALICIOUS user MODIFIES the role values IN the URL
3. RESUME STEP 4
4. ENDIF
```

Postcondition The MALICIOUS user has executed a function dedicated to another user with different role
Specific Alternative Threat Flow SATF2
RFS 4

```
1. IF the resource contains the role parameter in the HTTP post data THEN
2. The MALICIOUS user MODIFIES the role values IN the HTTP post data
3. RESUME STEP 4
4. ENDIF
```

Postcondition The MALICIOUS user has executed a function dedicated to another user with different role
Specific Alternative Flow SAF1
RFS 3

```
1. IF the response page contains the failure login message THEN
2. ABORT
3. ENDIF
```

Postcondition The MALICIOUS user CANNOT login
Specific Alternative Flow SAF2
RFS 6

```
1. IF the response page contains the error message THEN
2. ABORT
3. ENDIF
```

Postcondition The MALICIOUS user CANNOT execute a function dedicated to another user with different role
Mitigation Scheme Secure Coding for Web App

4. **Exploit HTTP verbs**

MISUSE CASE Exploit HTTP verbs
Description The misuse case simulates the OWASP testing activity OTG-INPVAL-003 (i.e., Testing for HTTP Verb Tampering) [Meucci and Muller, 2014]. As the HTML standard does not support request methods other than GET or POST, the MALICIOUS user crafts custom HTTP requests to test other methods (e.g., PUT, TRACE, OPTIONS, DELETE).
Precondition None
Primary Actor MALICIOUS user
Secondary Actors None
Dependency None
Generalization None
Threats Log In, Query Client Info, Create Account, Query Statistical Analysis Report
Assets system DATA
Basic Threat Flow

```
  1. DO
  2. The MALICIOUS user alters the HTTP request with FUZZ VALUES.
  3. The MALICIOUS user requests the url FROM the system.
  4. The system SENDS a response page TO the MALICIOUS user.
  5. The MALICIOUS user EXPLOITS the system.
  6. UNTIL the MALICIOUS user used all fuzz values
```

Postcondition The MALICIOUS user may exploit the system using different HTTP methods
Specific Alternative Flow SAF1

```
  RFS 5
  1. IF the response page contains the error message THEN
  2. RESUME STEP 6.
  3. ENDIF.
```

Postcondition The MALICIOUS user cannot exploit the system using different HTTP methods
Mitigation Scheme Secure Coding for Web App

5. **Exploit Insecure Direct Object References**

MISUSE CASE Exploit insecure direct object references
Description The misuse case simulates the OWASP testing activity OTG-AUTHZ-004 (Testing for Insecure Direct Object References) [Meucci and Muller, 2014]. This vulnerability allows the MALICIOUS user to bypass authorization and access resources directly by modifying the value of a parameter used to directly point to an object.
Precondition The malicious user needs to map out all locations in the application where user input is used to reference objects directly.
Primary Actor MALICIOUS user
Secondary Actors None
Dependency None
Generalization None
Threats Query Client Info, Create Account, Query Statistical Analysis Report
Assets client DATA
Basic Threat Flow
1. FOREACH url
2. DO
3. The MALICIOUS user SENDS FUZZ VALUES TO the system THROUGH the URL.
4. The system SENDS a response page TO the MALICIOUS user.
5. The MALICIOUS user EXPLOITS the system USING the url.
6. UNTIL the MALICIOUS user used all fuzz values
7. ENDFOR

Postcondition The MALICIOUS user may have accessed sensitive files or folders on the Web server.
Specific Alternative FlowSAF1
RFS 5
1.IF the response page contains the error message THEN
2. RESUME STEP 6.
3. ENDIF
Postcondition The MALICIOUS user cannot access sensitive files or folders on the Web server.
Mitigation Scheme Anonymize User Data

6. **Exploit Weak Lock Out Mechanism - Account**

MISUSE CASE Exploit weak lock out mechanism - Account
Description The misuse case simulates the OWASP testing activity OTG-AUTHN-003 (i.e., Testing for Weak lock out mechanism) [Meucci and Muller, 2014]. This test aims to verify if the account lockout mechanism mitigates brute force password guessing.
Precondition At least one client account has already been created in the system.
Primary Actor MALICIOUS user
Secondary Actors None
Dependency None
Generalization None
Threats Log In
Assets system DATA
Basic Threat Flow
1. DO
2. The MALICIOUS user SENDS DICTIONARY VALUES TO the system THROUGH the login page IN the username and password fields.
3. The system SENDS the response page TO the MALICIOUS user.
4. UNTIL the MALICIOUS user reaches the predefined threshold attempts.
5. The MALICIOUS user EXPLOITS the system.
Postcondition The MALICIOUS user knows the number of failed login attempts after which the account is locked.
Specific Alternative Threat Flow SATF1
RFS 4
1. IF the MALICIOUS user has been logged into the system THEN
2. The MALICIOUS user resets the attempt counter.
3. RESUME STEP 2.
4. ENDIF.
Postcondition The MALICIOUS user knows a valid account of the system.
Specific Alternative Flow SAF1
RFS 4
1. IF the response page contains a locked account message THEN
2. ABORT.
3. ENDIF.
Postcondition The system applies a lock out mechanism.
Mitigation Scheme Secure Coding for Web App

7. **Exploit Weak Lock Out Mechanism - IP**

MISUSE CASE Exploit weak lock out mechanism - IP
Description The misuse case simulates the OWASP testing activity OTG-AUTHN-003 (i.e., Testing for Weak lock out mechanism) [Meucci and Muller, 2014]. This test aims to verify if the IP lockout mechanism mitigates brute force password guessing.
Precondition At least one client account has already been created in the system.
Primary Actor MALICIOUS user
Secondary Actors None
Dependency None
Generalization None
Threats Log In
Assets system DATA
Basic Threat Flow
1. DO
2. The MALICIOUS user SENDS DICTIONARY VALUES TO the system THROUGH the login page IN the username and password fields.
3. The system SENDS the response page TO the MALICIOUS user.
4. UNTIL the MALICIOUS user reaches the high predefined number of attempts.
5. The MALICIOUS user EXPLOITS the system.
Postcondition The MALICIOUS user knows the number of failed login attempts after which the account is locked.
Specific Alternative Threat Flow SATF1
RFS 4
1. IF the MALICIOUS user has been logged into the system THEN
2. The MALICIOUS user resets the attempt counter.
3. RESUME STEP 2.
4. ENDIF.
Postcondition The MALICIOUS user knows a valid account of the system.
Specific Alternative Flow SAF1
RFS 4
1. IF the response page contains a locked account message THEN
2. ABORT.
3. ENDIF.
Postcondition The system applies a lock out mechanism for IP address.
Mitigation Scheme Secure Coding for Web App

8. **Exploit Weak Password Policy**

MISUSE CASE Exploit weak password policy
Description The misuse case simulates the OWASP testing activity OTG-AUTHN-007 (i.e., Testing for Weak password policy) [Meucci and Muller, 2014]. This test aims to evaluate the length, complexity, reuse and aging requirements of passwords.
Precondition At least one client account has already been created in the system.
Primary Actor MALICIOUS user
Secondary Actors None
Dependency None
Generalization None
Threats Reset Password
Assets system DATA
Basic Threat Flow
1. FOREACH PasswordType
2. The MALICIOUS user SENDS the password and the password confirmation value TO the system THROUGH the reset password page.
3. The system SENDS the response page TO the MALICIOUS user.
4. The MALICIOUS user EXPLOITS the system USING the PasswordType.
Postcondition The MALICIOUS user exploits the weak password policy.
Specific Alternative Flow SAF1
RFS 4
1. IF the response page contains a password mismatch message THEN
2. ABORT.
3. ENDIF.
Postcondition The password and the confirm password do not match.
Specific Alternative Flow SAF2
RFS 4
1. IF the response page contains a failure message THEN
2. ABORT.
3. ENDIF.
Postcondition The password does not match the password complexity policy.
Mitigation Scheme Secure Coding for Web App

9. **Exploit Weak Password Reset Functionality**

MISUSE CASE Exploit weak password reset functionality
Description The misuse case simulates the OWASP testing activity OTG-AUTHN-009 (i.e., Testing for weak password change or reset functionalities) [Meucci and Muller, 2014]. This misuse case aims to verify if the password reset functionality could be interrupted.
Precondition At least one client account has already been created in the system.
Primary Actor MALICIOUS user
Secondary Actors Client, Network sniffing tool
Dependency None
Generalization None
Threats Reset password
Assets client DATA
Basic Threat Flow

1. The MALICIOUS user RUNS the network sniffing tool.
2. The client SENDS the password and the password confirmation value TO the system THROUGH the reset password page.
3. The MALICIOUS user GETS the packets FROM the network sniffing tool.
4. The MALICIOUS user MODIFIES the new password fields IN the HTTP post data.
5. The MALICIOUS user RESENDS the modified packet TO the system.
6. The system SENDS the response page TO the MALICIOUS user.
7. The MALICIOUS user EXPLOITS the system.

Postcondition The MALICIOUS user successfully resets a client's password.

Specific Alternative Flow SAF1
RFS 7

1. IF response page contains a failure message THEN
2. ABORT.
3. ENDIF.

Postcondition The MALICIOUS cannot reset client's password.

Mitigation Scheme Anonymize User Data

10. **Get Credentials Transported over an Unencrypted Channel**

MISUSE CASE Get credentials transported over an unencrypted channel
Description The misuse case simulates the OWASP testing activity OTG-AUTHN-001 (i.e., Testing for Credentials Transported over an Encrypted Channel) [Meucci and Muller, 2014]. This misuse case verifies if the user's authentication data are transferred via an encrypted channel to avoid being intercepted by malicious users.
Precondition At least one client account has already been created in the system. The web browser has been configured to forward all packets to a proxy under the control of the MALICIOUS user.
Primary Actor MALICIOUS user
Secondary Actors None
Dependency None
Generalization None
Threats Log In
Assets client's credentials DATA
Basic Threat Flow

1. The MALICIOUS user SENDS the username and the password TO the system THROUGH the login page.
2. The system SENDS the response page TO the MALICIOUS user.
3. The MALICIOUS user MODIFIES the protocol IN the URL.
4. The MALICIOUS user RESENDS the modified packet TO the system.
5. The system SENDS the response page TO the MALICIOUS user.
6. The MALICIOUS user EXPLOITS the system USING the protocol.

Postcondition The MALICIOUS user knows that the client's account information is transferred on an unsecured channel.
Specific Alternative Threat Flow SATF1
RFS 4
1. IF the MALICIOUS user uses the GET method THEN
2. The MALICIOUS user EXPLOITS the system.
2. ABORT.
3. ENDIF.
Postcondition condition
Specific Alternative Flow SAF1
RFS 3
1. IF the response page contains a failed login message THEN
2. ABORT.
3. ENDIF.
Postcondition The MALICIOUS user filled wrong account credential or accessed wrong login page.
Specific Alternative Flow SAF2
RFS 5
1. IF the response page contains a failed login message THEN
2. ABORT.
3. ENDIF.
Postcondition The MALICIOUS user cannot log in the system.
Mitigation Scheme Anonymize User Data

11. **Get Default Credentials**

MISUSE CASE Get default credentials
Description The misuse case simulates the OWASP testing activity OTG-AUTHN-002 (i.e., Testing for default credentials) [Meucci and Muller, 2014]. The objective of this misuse case is to check if there exist default credentials in the system.
Precondition At least one client account has already been created in the system.
Primary Actor MALICIOUS user
Secondary Actors None
Dependency None
Generalization None
Threats Log In
Assets system DATA
Basic Threat Flow
1. DO
2. The MALICIOUS user SENDS DICTIONARY VALUES TO the system THROUGH the login page IN the username and password fields
3. The system SENDS the response page TO the malicious user
4. UNTIL the MALICIOUS user has been logged into the system
5. The MALICIOUS user EXPLOITS the system.
Postcondition The MALICIOUS user has been logged into the system by using a default credential.
Specific Alternative Flow SAF1
RFS 4
1. IF the response page contains a timeout message THEN
2. ABORT.
3. ENDIF.
Postcondition The MALICIOUS user has not been logged into the system
Specific Alternative Flow SAF2
RFS 4
1. IF the MALICIOUS user reaches the high predefined threshold attempts THEN
2. ABORT.
3. ENDIF.
Postcondition The MALICIOUS user has not been logged into the system
Mitigation Scheme Secure Coding for Web App

12. **Guess User Account**

MISUSE CASE Guess user account
Description The misuse case simulates the OWASP testing activity OTG-IDENT-004 (i.e., Testing for Account Enumeration and Guessable User Account) [Meucci and Muller, 2014]. The MALICIOUS user collects a set of valid usernames in the system.
Precondition At least one client account has already been created in the system.
Primary Actor MALICIOUS user
Secondary Actors None
Dependency None
Generalization None
Threats Log In
Assets client credentials DATA
Basic Threat Flow

```
  1. FOREACH combination
  2. The MALICIOUS user SENDS the username and the password TO the system THROUGH the login page.
  3. The system SENDS the response message TO the MALICIOUS user.
  4. The MALICIOUS user EXPLOITS the system USING the username and the password.
  5. ENDFOR.
```

Postcondition The MALICIOUS tried all the username and password combinations

Specific Alternative Threat Flow SATF1

```
  RFS 4
  1. IF the response page contains the wrong password message THEN
  2. The MALICIOUS user EXPLOITS the system USING the username.
  3. EXIT.
  4. ENDIF.
```

Postcondition The MALICIOUS user knows that the username exists but the password is wrong.

Specific Alternative Flow SAF1

```
  RFS 4
  1. IF the response page contains the unknown combination message THEN
  2. ABORT.
  3. ENDIF.
```

Postcondition The MALICIOUS user does not know whether the username exists or not.

Mitigation Scheme Secure Coding for Web App

A.2 Specifications Dedicated to the EDLAH2 System

1. Reuse Invitation to Create a User Account

MISUSE CASE Reuse invitation to create a user account
Description The misuse case aims to detect a vulnerability of EDLAH2 system [EDLAH2, 2017a]. The MALICIOUS user captures an invitation to create a new user account and then reuse it to create another user account.
Precondition The client user has the right to send an invitation to other people.
Primary Actor MALICIOUS user
Secondary Actors Client
Dependency None
Generalization None
Threats Invite user
Assets system DATA
Basic Threat Flow

1. The client SENDS the username and the password TO the system THROUGH the login page.
2. The MALICIOUS user RUNS the network sniffing tool.
3. The client SENDS the invitation request TO the system THROUGH the invitation page.
4. The MALICIOUS user GETS the packets FROM the network sniffing tool.
5. The MALICIOUS user MODIFIES the email field, and the recipient field, and the message field IN the HTTP post data.
6. The MALICIOUS user RESENDS the modified packet TO the system.
7. The system SENDS the response page TO the MALICIOUS user.
8. The MALICIOUS user EXPLOITS the system.

Postcondition The MALICIOUS user successfully creates a new account on the EDLAH2 system.

Specific Alternative Flow SAF1

RFS 8

1. IF response page contains a failure message THEN
2. ABORT.
3. ENDIF.

Postcondition The MALICIOUS cannot create a new account.

Mitigation Scheme Secure Coding for Web App

A.3 Mitigation Schemes

1. Secure Coding for Web App

Scheme Name	Secure Coding for Web App.
Brief Description	This mitigation guideline provides secure coding guidelines for the developers who develop the web app.
Actors	Software Developer, Security Engineer.
Mitigated Misuse Cases	Access Directory Traversal, Bypass Authentication Schema, Bypass Authorization Schema, Exploit HTTP verbs, Exploit Weak Lock Out Mechanism - Account, Exploit Weak Lock Out Mechanism - IP, Exploit Weak Password Policy, Get Default Credentials, Guess User Account, Reuse Invitation to Create a User Account
Compliance	ISO/IEC 27001:2013 clause A.6.1.5: Information security in project management, clause A.9.2: User access management, clause A.9.4: System & application access control, clause A.9.3.1: Use of secret authentication information.
Mitigation Tasks	1 Parameterize SQL queries, i.e., bind variables in stored procedures or prepared statements for SQL queries. Avoid dynamic SQL queries. 2 Avoid using user inputs in HTML outputs such as JavaScript and event handlers. If it cannot be avoided, sanitize user inputs using adequate security APIs (e.g. Apache's StringEscapeUtils) 3 Implement user authorization method. For example, the system prompts or alerts the client before launching sensitive actions (invoking sensitive APIs/resources, propagating sensitive information to external entities). 4 Implement user authentication method. For example, for every service request on the server side, attach an authentication token. Implement persistent authentication as opt-in rather than by default. 5 Implement a strong password policy that ensures password length, complexity, reuse and aging. 6 Implement the lockout mechanism that provides a balance between protecting accounts from unauthorized access and protecting users from being denied authorized access. 7 Eliminate all default credentials, default configuration in the system before deploying. 8 Educate users about phishing attacks. For example, educate them not to trust seemingly benign but malicious URLs sent to emails, messages, or social networking websites by unknown entities.

2. Anonymize User Data

Scheme Name	Anonymize User Data.
Brief Description	This mitigation guideline provides the guidelines for anonymizing user data to prevent external entities from being able to identify clients individually.
Actors	Software Developer
Mitigated Misuse Cases	Exploit Insecure Direct Object References, Exploit Weak Password Reset Functionality, Get Credentials Transported over an Unencrypted Channel
Compliance	ISO/IEC 27001:2013 clause A.6.1.5: Information security in project management, clause A.9.4: System & application access control, clause A.10.1: Cryptographic controls.
Mitigation Tasks	1 Store all user data in a secure, reliable database with proper access control rights specified. 2 Transmit user data between the system side and client side (e.g., browsers) via standard secure transport protocols (e.g. SSL/TLS). 3 Encrypt all explicit user identifiers (e.g., names, IDs, and addresses).

Appendix B

Catalog of Metamorphic Security Relations

In this chapter, we present 22 system-agnostic metamorphic security relations (see Section 6.7). These MRs are also available on the MST toolset website at: **`https://sntsvv.github.io/SMRL/`**.

These 22 MRs are derived from the following 16 OWASP testing activities [Meucci and Muller, 2014]:

1. Testing for Credentials Transported over an Encrypted Channel: OTG_AUTHN_001.
2. Testing for Bypassing Authentication Schema : OTG_AUTHN_004.
3. Testing for Weaker Authentication in Alternative Channel: OTG_AUTHN_010.
4. Testing Directory traversal/file include: OTG_AUTHZ_001a and OTG_AUTHZ_001b.
5. Testing for Bypassing Authorization Schema: OTG_AUTHZ_002, OTG_AUTHZ_002a, OTG_AUTHZ_002b, OTG_AUTHZ_002c, OTG_AUTHZ_002d, and OTG_AUTHZ_002e.
6. Testing for Privilege Escalation: OTG_AUTHZ_003.
7. Testing for Insecure Direct Object References: OTG_AUTHZ_004.
8. Test Number of Times a Function Can be Used Limits: OTG_BUSLOGIC_005.
9. Test HTTP Strict Transport Security: OTG_CONFIG_007.
10. Testing for Weak Encryption: OTG_CRYPST_004.
11. Testing for HTTP Verb Tampering: OTG_INPVAL_003.
12. Testing for HTTP Parameter pollution: OTG_INPVAL_004.
13. Testing for Session Fixation: OTG_SESS_003.
14. Testing for Logout Functionality: OTG_SESS_006.
15. Test Session Timeout: OTG_SESS_007.
16. Testing for Session puzzling: OTG_SESS_008.

```
import static smrl.mr.language.Operations.*;
import smrl.mr.language.Action;

package smrl.mr.owasp {
/*** A login operation should not succeed if performed on the HTTP channel.
 *
 * The 1st parameter of the operator IMPLIES is a boolean expression with
three clauses joined with logical conjunctions.
 * The 1st clause checks if the current action performs a log in. The 2nd
clause defines the follow-up input. The 3rd clause changes the channel of the
login action in the follow-up input.
 *
 * The 2nd parameter of IMPLIES checks if the output generated by the login
operation is different in the two cases.
 */
MR OTG_AUTHN_001 {
 {
   for ( Action action : Input(1).actions() ) {
      var pos = action.getPosition();
      IMPLIES(
         isLogin(action)                    //1st par (1st clause)
         && EQUAL ( Input(2), Input(1) )    //1st par (2nd clause)
         &&Input(2).actions.get(action.position).setChannel("http")
                                            //1st par (3rd clause)

         ,
         different ( Output(Input(1), pos), Output(Input(2), pos) )
                                       //2nd par of IMPLIES
      );//end-IMPLIES
   }//end-for
 }}//end-MR
}//end-package
```

Figure B.1. MR OTG_AUTHN_001

```
import static smrl.mr.language.Operations.*;
import smrl.mr.language.Action;

package smrl.mr.owasp {
/*** Without being authenticated, a user should not be able to access a page
that normally can be reached only through the user interface of authenticated
users.
 *
 * The 1st parameter of the operator IMPLIES is a boolean expression with
three clauses joined with logical conjunction.
 * The 1st clause checks if the current action has been performed after a log
in. The 2nd clause checks if the current action has ever been performed by a
non-authenticated user. The 3rd clause defines a follow-up input that performs
only the given action, without logging in before.
 *
 * The 2nd parameter of IMPLIES checks if the output generated by the action
is different in the two cases.
 */
MR OTG_AUTHN_004 {
 {
   for ( Action action : Input(1).actions() ) {
      IMPLIES(
         afterLogin( action )    //1st par of IMPLIES (1st clause)
         && notVisibleWithoutLoggingIn( action.getUrl() )
                                 //1st par of IMPLIES (2nd clause)
         && EQUAL( Input(2), action )
                                 //1st par of IMPLIES (3rd clause)

         ,
         different(
            Output(Input(1), action.position ),
            Output(Input(2), action.position ) )
                                 //2nd par of IMPLIES
      );//end-IMPLIES
   }//end-for
 }}//end-MR
}//end-package
```

Figure B.2. MR OTG_AUTHN_004

```
import static smrl.mr.language.Operations.*;
import smrl.mr.language.Action;

package smrl.mr.owasp{
/*** Without begin authenticated, a user should not be able to access a page
that normally can be reached only through the user interface of authenticated
users.
 * This should be true even if she tries on the http channel (i.e., the result
of a same operation being performed on a different channel should be
different).
 *
 * The 1st parameter of the operator IMPLIES is a boolean expression with four
clauses joined with logical conjunction.
 * The 1st clause checks if the current action has been performed after a
login. The 2nd clause checks if the current action is not occurring already on
the http channel. The 3rd clause defines a follow-up input. The 4th clause set
the channel of the action the follow-up input to "http".
 *
 * The 2nd parameter of the operator IMPLIES checks if the output generated by
the action is different in the two cases.
 */
MR OTG_AUTHN_010 {
 {
    for ( Action action : Input(1).actions() ){
        var pos = action.position;
        IMPLIES(
          afterLogin( action )    //1st par of IMPLIES (1st clause)
          &&!Input(1).actions().get(pos).getChannel().equals("http")
                                 //1st par of IMPLIES (2st clause)
          && EQUAL ( Input(2), Input(1) )
                                 //1st par of IMPLIES (3nd clause)
          && Input(2).actions().get(pos).setChannel("http")
                                 //1st par of IMPLIES (4rd clause)
          ,
          different( Output(Input(1),pos), Output(Input(2),pos) )
                                 //2nd par of IMPLIES
      );//end-IMPLIES
    }//end-for
 }}//end-MR
}//end-package
```

Figure B.3. MR OTG_AUTHN_010

```
import static smrl.mr.language.Operations.*;
import smrl.mr.language.Action;

package smrl.mr.owasp{
/*** A file path passed in a parameter should never enable a user to access
data that is not provided by the user interface.
 * This metamorphic relation contains two nested loops; the first iterates
over the actions in the input sequence, the second iterates over the
parameters of the action.
 *
 * The 1st parameter of the operator IMPLIES is a boolean expression with two
clauses joined with a logical conjunction.
 * The 1st clause defines a follow-up input that is a copy of the source
input. The 2nd clause sets the value of a parameter to a random file path.
 *
 * The 2nd parameter of IMPLIES verifies the result. It is implemented as an
OR operation where the 1st parameter verifies that the follow-up input leads
to an error page.
 * The 2nd parameter deals with the case in which the generated request is
valid, and verifies that the returned content is something that the user has
the right to access.
 */
MR OTG_AUTHZ_001a {
 {
   for ( Action action : Input(1).actions() ){
      for (var par=0; par < action.getParameters().size(); par++){
         var pos = action.getPosition();
         IMPLIES(
            EQUAL( Input(2), Input(1) )
                           //1st par of IMPLIES (1st clause)
            && Input(2).actions().get(pos)
                .setParameterValue(par, RandomFilePath())
                           //(2nd clause)
            ,
            //2nd par of IMPLIES, OR operator receiving 2 parameters
            OR(
               Output(Input(2),pos).isError()    //1st par of OR
               ,
               userCanRetrieveContent(
                      action.getUser(),
                      Output(Input(2),pos)) )    //2nd par of OR
         );//end-IMPLIES
      }//end-for
   }//end-for
 }}//end-MR
}//end-package
```

Figure B.4. MR OTG_AUTHZ_001a

```
import static smrl.mr.language.Operations.*;
import smrl.mr.language.Action;

package smrl.mr.owasp{
/*** A file path passed in the URL of a request should never enable a user to
access data that is not provided by the user interface.
 * This metamorphic relation contains two nested loops; the first is used to
introduce relative paths in the query (jumps to parent folders), the second
iterates over the actions in the input sequence.
 *
 * The 1st parameter of the operator IMPLIES is a boolean expression with
three clauses joined with a logical conjunction.
 * The 1st clause verifies whether the current action has been not performed
by an administrator. The 2nd clause checks if the current action has been
performed after a login. The 3rd clause defines a follow-up input that is a
copy of the source input. The 4th clause adds to the end of the current URL a
relative path to a file. The 5th clause verifies that the given path was not
tried in a previous execution of the loop (to speed up).
 *
 * The 2nd parameter of IMPLIES verifies the result. It is implemented as an
OR operation where the 1st parameter verifies that the follow-up input does
not lead to a file; the 2nd parameter deals with the case in which the
generated request is valid, and verifies that the returned file is something
that the user has the right to access; the 3rd parameter verifies that the
follow-up input leads to an error page.
*/
MR OTG_AUTHZ_001b {
 {
   var sep="/";
      for ( var par=0; par < 4; par++ ){
         for ( Action action : Input(1).actions() ){
            var pos = action.getPosition();
            var newUrl = action.urlPath+sep+RandomFilePath();
            IMPLIES(
               //1st clause of IMPLIES
               !isAdmin(action.user) && afterLogin(action) &&
               EQUAL( Input(2), Input(1) ) &&
               Input(2).actions().get(pos).setUrl( newUrl ) &&
               notTried( action.getUser(), newUrl )
               , //2nd par of IMPLIES
               TRUE (
                  Output(Input(2),pos).noFile() ||
                  userCanRetrieveContent( action.getUser(),
                                 Output(Input(2),pos).file()) ||
                  Output(Input(2),pos).isError()
            );//end-IMPLIES
         }//end-for
         sep=sep+"../";
      }//end-for
   }}//end-MR
}//end-package
```

Figure B.5. MR OTG_AUTHZ_001b

```
import static smrl.mr.language.Operations.*
import smrl.mr.language.Action;

package smrl.mr.owasp {
   /*** A URL that cannot be reached by a user while navigating the user
interface should not be available to that same user even when she directly
requests the URL to the server.
    * For this reason, an input sequence that is valid for a given user, should
not lead to the same output when it is executed by another user, if it
includes access to a URL with these characteristics.
    * The metamorphic relation iterates over all the actions of an input
sequence.
    *
    * The 1st parameter of IMPLIES is made of three clauses.
    * The 1st clause checks whether the user in User() is not a supervisor of
the user performing the current action. The 2nd clause verifies that the user
cannot retrieve the URL of the action through the GUI (based on the data
collected by the crawler). The 3rd clause defines a follow-up input that
matches the source input except that the credentials of User() are used in
this case.
    *
    * The 2nd parameter of IMPLIES verifies the result. It is implemented as an
OR operation where the 1st parameter verifies that the follow-up input leads
to an error page; The 2nd parameter verifies that the output generated by the
action containing the URL indicated above leads to two different outputs in
the two cases.
    */
MR OTG_AUTHZ_002 {
 {
   for ( Action action : Input(1).actions() ){
      IMPLIES(
         //1st par of IMPLIES
         (!isSupervisorOf(User(), action.user)) &&
         cannotReachThroughGUI( User(), action.url ) &&
         EQUAL( Input(2), changeCredentials(Input(1), User()) )
         ,
         OR(     //2nd par of IMPLIES
            isError(Output(Input(1),action.position)),
            NOT(Output(Input(1),action.position).equals(
               Output(Input(2),action.position)))
      )); //end-IMPLIES
   } //end-for

 }} //end-MR
}//end-package
```

Figure B.6. MR OTG_AUTHZ_002

```
import static smrl.mr.language.Operations.*
import smrl.mr.language.actions.ClickOnNewRandomElement
import smrl.mr.language.Action;

package smrl.mr.owasp {
/*** If a redirecting URL cannot be reached by a user while navigating the
user interface, the same URL, if directly requested to the server, should
not enable the same user to access a page where the click on one of its
elements (e.g., a warning message) enables the user to access the content of
the URL.
 * The metamorphic relation iterates over all the actions of an input
sequence.
 *
 * The 1st parameter of IMPLIES is made of three clauses.
 * The 1st clause checks whether the user in User() is not a supervisor of the
user performing the current action. The 2nd clause verifies that the user
cannot retrieve the URL of the action through the GUI (based on the data
collected by the crawler). The 3rd clause defines a follow-up input that
matches the source input except that the credentials of User() are used in
this case.
 *
 * The 2nd parameter of IMPLIES verifies the result. It is made of three
clauses.
 * The 1st clause verifies that the original URL does not perform any
redirect. The 2nd clause verifies that the original URL does not perform any
redirect. The 3rd clause verifies that the follow up input does not lead to
the same redirect from the original input.
 */
MR OTG_AUTHZ_002a {
 {
   for ( Action action : Input(1).actions() ){
      var pos = action.getPosition();
      IMPLIES(
         (!isSupervisorOf(User(), action.user)) && // 1st par
         cannotReachThroughGUI(User(), action.url) &&
         EQUAL( Input(2), changeCredentials( Input(1), User() ) )
         ,
         ( Output(Input(1), pos).redirectURL()===null ||
           Output(Input(2), pos).redirectURL()===null ) ||
           NOT(
            EQUAL (
               Output(Input(2), pos).redirectURL(),
               Output(Input(1), pos).redirectURL()))
      ); //end-IMPLIES
   }//end-IMPLIES
 }} //end-MR
} //end-package
```

Figure B.7. MR OTG_AUTHZ_002a

```
import static smrl.mr.language.Operations.*
package smrl.mr.owasp {
   /*** If a certain action is not available to a given user, this user should not
be able to perform the action. Assume we have two users, user a and user b. Given
(1) a source input as a sequence of actions performed by a user 'a' which contains
an action y that is dedicated to user a (i.e., it is not visible to user b) and
(2) a follow-up input that is a copy of that sequence which, however, includes,
before action y, an action that matches action y (e.g., same URL requested) but is
performed by user 'b'.
   * The result of action y should not be different when performed in the source
input (i.e., without any action of b) or in the follow-up input (i.e., when
performed also by user b).
   * This MR contains two loops. The first iterates over the actions of the source
input to identify a login operation (action x) for user a, the second iterates
over the remaining y-th actions.
   * The 1st parameter of IMPLIES defines the follow-up input. The 1st clause
checks whether the user in User() is not a supervisor of the user performing the
y-th action. The 2nd clause checks that action y cannot be accessed by user b
(User()). The 3rd clause defines Input(2) which just performs a login. The 4th
clause defines Input(3) which just performs a login as user b. The 5th clause
creates a copy of Input(1) with a login as b before action y (this way action y is
performed as User b). The 6th clause adds after action y+1 (the original action y
now shifted) new copy of action y (now performed by user b). The 7th clause adds
after the new copy of action y a new login as user a.
   * The 2nd parameter of IMPLIES checks that the output of the action y in the
two sequences remains the same when performed by user a (in the follow-up sequence
the action of user a is shifted by three because three actions are introduced, the
login of user b, the current action and a new login for user a). */
MR OTG_AUTHZ_002b {
 {
   for(var x = 0; x < Input(1).actions().size() ; x++){
      for (var y = x+1;
         isLogin(Input(1).actions().get(x)) &&
          (y < Input(1).actions().size()); y++) {
         IMPLIES( //1st par of IMPLIES including 7 clauses
            (!isSupervisorOf(User(),Input(1).actions().get(y).user)) &&
            cannotReachThroughGUI( User(),
               Input(1).actions().get(y).getUrl()) &&
            EQUAL(Input(2), Input(1).actions().get(x)) &&
            EQUAL(Input(3), changeCredentials(Input(2), User())) &&
            EQUAL( Input(4), addAction(Input(1), y,
                   Input(3).actions().get(0))) &&
            EQUAL( Input(5), addAction(Input(4), y+1,
                   Input(1).actions().get(y))) &&
            EQUAL( Input(6), addAction(Input(5), y+2,
                   Input(1).actions().get(x) ) )
            , //2nd par of IMPLIES
            EQUAL(Output(Input(1), y), Output(Input(6), y+3 ))
         ); //end-IMPLIES
      } //end-for
   } //end-for
 }} //end-MR
} //end-package
```

Figure B.8. MR OTG_AUTHZ_002b

```
import static smrl.mr.language.Operations.*

package smrl.mr.owasp {
/*** A URL that cannot be reached by a user while navigating the user
interface should not be available to that same user even when she directly
requests the URL to the server.
   * The metamorphic relation iterates over all the actions of an input
sequence.
   *
   * The 1st parameter of IMPLIES is made of four clauses. The 1st clause
checks whether the user in User() is not a supervisor of the user performing
the y-th action. The 2nd clause verifies that the y-th action is performed
after a login. The 3rd clause verifies that the follow-up user cannot retrieve
the URL of the action through the GUI (based on the data collected by the
crawler). The 4th clause defines a follow-up input that performs the login as
the follow-up user and then performs the given action.
   * The 2nd parameter of IMPLIES verifies the result. It is implemented as an
OR operation where The 1st parameter checks if the y-th action from the source
input leads to an error page; The 2nd parameter verifies if the output
generated by the action containing the URL indicated above, lead to two
different outputs in the two cases.
   */
MR OTG_AUTHZ_002c {
 {
   for(var y = Input(1).actions().size()-1; ( y > 0 ); y--){
      IMPLIES( //1st par of IMPLIES including 4 clauses
          (!isSupervisorOf( User(),
                 Input(1).actions().get(y).user)) &&
         afterLogin(Input(1).actions().get(y)) &&
         cannotReachThroughGUI( User(),
            Input(1).actions().get(y).getUrl()) &&
         EQUAL(
            Input(2),
            Input(LoginAction(User()), Input(1).actions().get(y)))
         , //2nd par of IMPLIES
         OR(
            isError(Output(Input(1), y)),
            different( Output(Input(1), y), Output(Input(2), 1))
         )
      ); //end-IMPLIES
   } //end-for
 }} //end-MR
} //end-package
```

Figure B.9. MR OTG_AUTHZ_002c

```
import static smrl.mr.language.Operations.*
package smrl.mr.owasp {
 /*** A user should not be able to overwrite an admin file by writing its path
in a file form.
 * The first loop iterates over all the actions of an input sequence.
 * The second loop looks for an action that contains a form that appear to be
used to specify paths.
 * The 1st parameter of IMPLIES is made of four clauses. The 1st clause
verifies that the follow-up user is not an admin (admin may access any file).
The 2nd clause verifies that the selected text input in a form contains a file
path (or a file name). The 3rd clause verifies defines a follow-up input that
is a copy of the source input. The 4th clause puts a randomly selected path of
an admin file in the selected form input of the follow-up sequence.
 * The 2nd parameter of IMPLIES verifies the result. It is implemented as an
OR operation where the 1st parameter checks if the y-th action from the source
input leads to an error page; The 2nd parameter verifies if the output
generated by sequence containing the path to the admin file is different than
the output of the sequence performed by the original user. We do not check
only for the output of action x because the error might be observed afterwards
(e.g., during execution). */
MR OTG_AUTHZ_002d {
 {
   for ( var x=0; Input(1).containFormInputForFilePath() &&
         x<Input(1).actions().size; x++) {
      var action = Input(1).actions.get(x);
      var randomPath = RandomAdminFilePath();
      var formInputs = action.getFormInputs();
      for(var i=0; action.containFormInputForFilePath() &&
            i< formInputs.size(); i++){
         var formInput =
               action.getFormInputs().get(i).getAsJsonObject();
         IMPLIES( //1st par of IMPLIES including 4 clauses
            ! isAdmin(action.user) &&
            isFormInputForFilePath( formInput ) &&
            EQUAL(Input(2), Input(1)) &&
            updateStringFormInput(
               Input(2).actions.get(x).getFormInputs()
                  .get(i).getAsJsonObject(),
               randomPath)
            , //2nd par of IMPLIES
            OR(
               isError(Output(Input(1))),
               different(Output(Input(1)), Output(Input(2))
            )
         ); //end-IMPLIES
      } //end-for
   } //end-for
 }} //end-MR
} //end-package
```

Figure B.10. MR OTG_AUTHZ_002d

```
import static smrl.mr.language.Operations.*
package smrl.mr.owasp {
/*** This MR matches OTG_AUTHZ_002b with the difference that in this case we
check for the content provided by action (y+1).
 *
 * The MR verifies that the content provided to the original user is either an
error or is not anomalous (i.e., was already observed).
 */
MR OTG_AUTHZ_002e {
 {
   for(var x = 0; (x < Input(1).actions().size() ); x++){
      for (var y = x+1;
         (isLogin(Input(1).actions().get(x)) &&
         y < Input(1).actions().size()); y++) {
         IMPLIES( //1st par of IMPLIES including 5 clauses
          (!isSupervisorOf(User(), Input(1).actions().get(y).user)) &&
         cannotReachThroughGUI(User(),
            Input(1).actions().get(y).getUrl()) &&
         EQUAL( Input(2), Input(1).actions().get(x)) &&
         EQUAL( Input(3), changeCredentials(Input(2), User())) &&
         EQUAL( Input(4),
            addAction(Input(1), y, Input(3).actions().get(0)))
         , //2nd par of IMPLIES
         OR(
            userCanRetrieveContent(
               Input(4).actions().get(y+1).getUser(),
               Output(Input(4), y+1))
            ,
            Output(Input(4), y+1).isError()
         )
      ); //end-IMPLIES
      } //end-for
   } //end-for
 }} //end-MR
} //end-package
```

Figure B.11. MR OTG_AUTHZ_002e

```
import static smrl.mr.language.Operations.*;
import smrl.mr.language.Action;

package smrl.mr.owasp{
/*** If an action is not expected to be executed by a user 'u' (because is not
available in his GUI), then user 'u' should not be able to execute that action
even if he changes the user id parameter in that action (i.e., the action
should lead to different results when executed by a valid and invalid user).
 *
 * The first loop iterates over all the actions of the input sequence.
 * The second iterates over all the parameters of the action to identify a
parameter that specifies the user id.
 *
 * The 1st parameter of the operator IMPLIES is a boolean expression with four
clauses joined with logical conjunctions.
 * The 1st clause checks if the current action contains a user ID.
 * The 2nd clause defines the follow-up input as a copy of the source input.
 * The 3rd clause changes the user ID to the one of User
 * The 4th clause changes the login credentials of the follow-up input to the
one of User
 *
 * The 2nd parameter of IMPLIES checks if the output generated by the action
is different in the two cases.
 */
MR OTG_AUTHZ_003 {
{
   for ( Action action : Input(1).actions() ){
      for (var par=0; par < action.getParameters().size(); par++ ){
         var pos = action.getPosition();
         IMPLIES (
            isUserIdParameter(action,par,action.getUser() ) &&
            ( equal ( Input(2), Input(1) ) &&
            Input(2).actions().get(pos)
               .setParameterValue(par,User()) ) &&
            equal (Input(3), changeCredentials(Input(1), User()) )
            ,
            different (Output(Input(2),pos), Output(Input(3),pos)) )
      }
   }
 }
}}
```

Figure B.12. MR OTG_AUTHZ_003

```
import static smrl.mr.language.Operations.*;
import smrl.mr.language.Action;

package smrl.mr.owasp{
/*** By randomly changing the parameter values passed to URLs, a user should
not be able to retrieve content she cannot retrieve from GUI.
 *
 * The first loop iterates over all the actions of the input sequence.
 * The second iterates over all the parameters of the action.
 *
 * The 1st parameter of the operator IMPLIES is a boolean expression with two
clauses joined with logical conjunctions.
 * The 1st clause defines the follow-up input.
 * The 2nd clause set a parameter value to a random value.
 *
 * The 2nd parameter of IMPLIES checks if the content of the output generated
by the login operation is either an error message or some content that can be
retrieved from the GUI.
 */
MR OTG_AUTHZ_004 {
 {
   for ( Action action : Input(1).actions() ){
      for (var par=0; par < action.getParameters().size(); par++){
         var pos = action.getPosition();
         IMPLIES(
            EQUAL ( Input(2), Input(1) )
            &&  Input(2).actions().get(pos)
               .setParameterValue(par,
               RandomValue( typeOf( action.getParameterValue(par)))
            ,
            OR( Output( Input(2),pos).isError(),
                userCanRetrieveContent( action.user,
                  Output(Input(2),pos))
            )
         );//end-IMPLIES
      }//end-for
   }//end-for
 }}//end-MR
}
```

Figure B.13. MR OTG_AUTHZ_004

```
import static smrl.mr.language.Operations.*;
import smrl.mr.language.Action;

package smrl.mr.owasp{
/*** Some URLs are expected to be used only once. These URLs can be identified
(by the data collection framework) by checking if a same action (e.g.,
clicking on a button) triggers always different
(e.g., the button URL is always different) over different executions.
 * In this case a user should not be able to reuse the URL multiple times
(e.g., sending POST data to the same URL).
 * The loop iterates over all the actions of the input.
 *
 * The 1st parameter of the operator IMPLIES is a boolean expression with four
clauses joined with logical conjunctions. The 1st clause checks if the URL of
the current action changes over multiple executions. The 2nd clause defines
the follow-up input as a copy of the source input where the action above is
duplicated.
 *
 * The 2nd parameter of IMPLIES checks if the output generated by the second
action different than in the case of the first action.
 */
MR OTG_BUSLOGIC_005 {
 {
   for ( Action action : Input(1).actions() ){
      var pos = action.getPosition();
      IMPLIES( (
         urlOfActionChangesOverMultipleExecutions( action )
         && equal ( Input(2), addAction( Input(1), pos, action )))
         ,
         different( Output(Input(1),pos), Output(Input(2), pos) ) )
   }
 }}
}
```

Figure B.14. MR OTG_BUSLOGIC_005

```
import static smrl.mr.language.Operations.*;
import smrl.mr.language.Action;

package smrl.mr.owasp {
/*** An action with strict transport security header should not be available
on the http channel.
 * The loop iterates over all the actions of the input.
 *
 * The 1st parameter of the operator IMPLIES is a boolean expression with
three clauses joined with logical conjunctions.
 * The 1st clause defines the follow-up input.
 * The 2nd clause checks if the output of the source input has strict
transport security header.
 * The 3rd clause set the channel of the action to http
 *
 * The 2nd parameter of IMPLIES checks that if the modified action has not
been redirected to httpsthen the output generated by the action should be
different than in the case of the source input.
 */
MR OTG_CONFIG_007 {
 {
   for ( Action action : Input(1).actions() ) {
      var pos = action.getPosition();
      IMPLIES(
          ( equal ( Input(2) , Input(1) ) &&
         Output(Input(1),pos).hasStrictTransportSecurityHeader() &&
         Input(2).actions().get(pos).setMethod("http") )
         ,
         AND (
            equal ( Output(Input(2),pos).getChannel(), "https" ),
            equal ( Output(Input(1),pos) , Output(Input(2),pos))))
   }
 }}
}
```

Figure B.15. MR OTG_CONFIG_007

```
import static smrl.mr.language.Operations.*;
import smrl.mr.language.Action;

package smrl.mr.owasp {
/*** Weak encryption algorithms should not be available.
 *
 * The loop iterates over all the actions of the input.
 *
 * The 1st parameter of the operator IMPLIES is a boolean expression with
three clauses joined with logical conjunctions.
 * The 1st clause checks if the action works on the encrypted channel.
 * The 2nd clause defines a follow-up input.
 * The 3rd clause set the encryption algorithms to a weak one.
 *
 * The 2nd parameter of IMPLIES checks that the output generated by the action
using the weak encryption algorithm lead to different results.
 */
MR OTG_CRYPST_004 {
 {
   for ( Action action : Input(1).actions() ){
      IMPLIES (
          (isEncrypted( action ) &&
         equal ( Input(2) , Input(1) ) &&
         Input(2).actions().get(action.position)
               .setEncryption( WeakEncryption() ) )
         ,
         different ( Output( Input(1) ), Output( Input(2) ) ) )
   }
 }}
}
```

Figure B.16. MR OTG_CRYPST_004

```
import static smrl.mr.language.Operations.*;
import smrl.mr.language.Action;

package smrl.mr.owasp {
/*** This MR checks that  actions available with one HTTP method (e.g., POST )
should not be available with another method (e.g., DELETE).
 * The metamorphic relation iterates over all the actions of an input
sequence.
 *
 * The 1st parameter of IMPLIES is made of two clauses.
 * The 1st clause verifies that the user cannot retrieve the URL of the action
through the GUI (based on the data collected by the crawler).
 * The 2nd clause defines a follow-up input in which the selected action is
performed using a different HTTP method.
 *
 * The 2nd parameter of IMPLIES verifies that the output generated by the
modified action is different in the two cases.
 */
MR OTG_INPVAL_003 {
 {
   for ( Action action : Input(1).actions() ) {
      var pos = action.getPosition();
      IMPLIES(
         ( EQUAL( Input(2) , Input(1) ) &&
         Input(2).actions().get(pos).setMethod( HttpMethod() ))
         ,
         different ( Output(Input(1),pos),  Output(Input(2),pos) ))
   }
 }}
}
```

Figure B.17. MR OTG_INPVAL_003

```
import static smrl.mr.language.Operations.*;
import smrl.mr.language.Action;

package smrl.mr.owasp{
/*** Duplicating a parameter value should not lead to a different behaviour in
the system.
 * The first loop iterates over all the actions of an input sequence.
 * The second loop iterates over all the parameters.
 *
 * The 1st parameter of IMPLIES is made of two clauses.
 * The 1st clause defines a follow-up input.
 * The 2nd clause duplicates one parameter.
 *
 * The 2nd parameter of IMPLIES verifies that the output generated by the
modified action is the same in the two cases.
 */
MR OTG_INPVAL_004 {
 {
   for ( Action action : Input(1).actions() ){
      for (var par=0; par < action.getParameters().size(); par++ ){
         var pos = action.getPosition();
         IMPLIES (
            ( equal ( Input(2), Input(1) ) &&
            Input(2).actions().get(pos).addParameter(
                     action.getParameterName(par),
                     action.getParameterValue(par) ))
            ,
            equal ( Output(Input(1) ) , Output( Input(2) )))
      }
   }
 }}
}
```

Figure B.18. MR OTG_INPVAL_004

```
import static smrl.mr.language.Operations.*;
import smrl.mr.language.Action;

package smrl.mr.owasp{
/*** A login action performed by a user already authenticated should always
trigger the generation of a new session ID.
* This metamorphic relation contains two nested loops. the first iterates over
the inputs to find a sign up action, the second iterates over the actions that
follow the sign up. The second loop is necessary to check that a sign up
action repeated at any point of the action sequence leads to a new session ID.
 *
 * The 1st parameter of the operator IMPLIES is a boolean expression with two
clauses joined with logical conjunction.
 * The 1st clause checks if the current action has been performed after a
login.
 * The 2nd clause defines a follow-up input with the sign up action being
duplicated in a certain position.
 *
 * The 2nd parameter of IMPLIES checks if the session ID of the response page
sent after the two successive login actions is different.
 */
MR OTG_SESS_003 {
 {
   for( Action signup : Input(1).actions() ){
      for ( var i=0;
         isSignup(signup) && i < Input(2).actions().size; i++ ) {
         var f = Input(2).actions().get(i);
         var pos = f.getPosition();
         IMPLIES(
            afterLogin( f ) &&   //1st par of IMPLIES (1st clause)
            EQUAL(
               Input(3),
               addAction( Input(2), pos+1, signup ))  //(2nd clause)
            ,
            different(     //2nd par of IMPLIES
               Output(Input(3), pos).getSession(),
               Output(Input(3), pos+1).getSession())
         );//end-IMPLIES
       }//end-for
   }//end-for
 }}//end-MR
}
```

Figure B.19. MR OTG_SESS_003

```
import static smrl.mr.language.Operations.*;

package smrl.mr.owasp{
/*** A logout action should always lead to a new session.
 * This MR iterates over all the actions to find a logout action. The second
loop iterates over all the actions to find an action performed after login.
 *
 * The 1st parameter of the operator IMPLIES is a boolean expression with two
clauses joined with logical conjunction.
 * The 1st clause checks if the current action x is a logout operation.
 * The 2nd clause checks that the action y is performed after a login.
 * The 3rd clause checks that the action y is not a login.
 * The 4th clause defines a follow-up input with the logout action being
duplicated in position y.
 *
 * The 2nd parameter of IMPLIES checks if the session ID before and after
executing the logout is different.
 */
MR OTG_SESS_006 {
{
   for ( var x=0; x < Input(1).actions().size() ; x++ ){
      for ( var y=0; y < x ; y++ ){
         IMPLIES (
            isLogout( Input(1).actions().get(x) ) &&
            afterLogin( Input(1).actions().get(y) ) &&
            ! isLogin( Input(1).actions().get(y) ) &&
            EQUAL ( Input(2) , copyActionTo( Input(1), x, y ) )
            ,
            different(Session(Input(2),y-1), Session(Input(2),y))) ;
      }
   }
 }}
}
```

Figure B.20. MR OTG_SESS_006

```
import static smrl.mr.language.Operations.*;
import smrl.mr.language.Action;

package smrl.mr.owasp{
/*** After a session timeout the user should not be able to perform an action
that requires to be logged in.
 * This MR iterates over all the actions to find actions executed within a
session, after login.
 *
 * The 1st parameter of the operator IMPLIES is a boolean expression with
three clauses joined with logical conjunction.
 * The 1st clause checks that the action is generally not available without
login. The 2nd clause checks if the session is not null. The 3rd clause checks
that a session timeout is set. The 4th clause defines a follow-up input where
the selected action is executed after timeout (usually simulated).
 *
 * The 2nd parameter of IMPLIES checks if the output of the action generated
after timeout is different than in the case in which it is executed before the
timeout.
 */
MR OTG_SESS_007 {
 {
   for ( Action action : Input(1).actions() ){
      IMPLIES (
         notAvailableWithoutLoggingIn(action) &&
         NOT ( NULL ( action.session ) ) &&
         action.session.timeout > 0 &&
         EQUAL ( Input(2) ,
            addAction( Input( 1 ),
               action.position,
               Wait(action.session.timeout) ))
         ,
         different (
            Output( Input(1), action.position ),
            Output( Input(2), action.position ) ));
   }
 }}
}
```

Figure B.21. MR OTG_SESS_007

```
import static smrl.mr.language.Operations.*;
import smrl.mr.language.Action;

package smrl.mr.owasp{
/*** An action that (1) is available without logging in and (2) generates a
session, should not enable a user to execute an action that requires to be
logged in.
 * This MR iterates over all the actions of the input.
 *
 * The 1st parameter of the operator IMPLIES is a boolean expression with
three clauses joined with logical conjunction.
 * The 1st clause checks that the current action is not available without
being logged in. The 2nd clause looks for an action available without being
logged-in that generates a session. The 3rd clause defines a follow-up input
that executes two actions, the action available without being logged in, and
the selected action (i.e., the one available only by being logged-in).
 *
 * The 2nd parameter of IMPLIES checks that the output of the action is
different when execute with and without being logged in (even if after an
action that does not require a log-in but generates a session).
 */
MR OTG_SESS_008 {
{
   for ( Action action : Input(1).actions() ){
      IMPLIES(
         notAvailableWithoutLoggingIn( action ) &&
         NOT (NULL(ActionAvailableWithoutLogin().getSession() ) ) &&
         EQUAL( Input(2) ,
            Input( ActionAvailableWithoutLogin(), action ) )
         ,
         different (
            Output( Input(1), action.position ),
            Output( Input(2), 1 ) ) );
   }
 }}
}
```

Figure B.22. MR OTG_SESS_008

Appendix C

Analysis of Weaknesses Reported in the CWE Database

This chapter provides eight Tables with the data concerning the CWE weaknesses presented in Chapter 7.

Columns *'CWE ID'* and *'Weakness'* report the identification and the name of each weakness, which are numbered and named by CWE [CWE, 2020s]. The column *'Gen.'* indicates if the weakness is generic (see Section 7.2). The fourth and the fifth columns are related to MCP, while the sixth and the seventh columns are related to MST. The fourth and the sixth columns *'App.'* indicate if the weakness can be addressed by MCP and MST, respectively. The fifth and the seventh columns *'Conditions'* specify testability features (i.e., applicability conditions enables the application of our approaches) or reasons (i.e., inapplicability reasons preventing the application of our approaches to address the weaknesses) for MCP and MST, respectively.

Tables C.1 to C.5 group 223 weaknesses into 12 security design principles: *Audit, Authenticate Actors, Authorize Actors, Cross Cutting, Encrypt Data, Identify Actors, Limit Access, Limit Exposure, Lock Computer, Manage User Sessions, Validate Inputs,* and *Verify Message Integrity*. Meanwhile, 43 weaknesses related to OWASP Top 10 Security Risks are categorized in ten groups named A1 to A10. They are shown in Tables C.6 and C.7. Table C.8 presents weaknesses in the CWE Top 25 most dangerous software errors view.

Table C.1. Weaknesses related to CWE security design principles view - Audit & Authenticate Actors

CWE ID	Weakness	Gen.	MCP		MST	
			App.	Conditions	App.	Conditions
Audit - (1009)						
117	Improper Output Neutralization for Logs	1	1	TF1 + TF10	-	R4
223	Omission of Security-relevant Information	1	-	R2	-	R2
224	Obscured Security-relevant Information by Alternate Name	1	1	TF1 + TF2	-	R5
532	Inclusion of Sensitive Information in Log Files	1	1	TF1 + TF2	-	R5
778	Insufficient Logging	1	1	TF1 + TF10	-	R4
779	Logging of Excessive Data	1	1	TF1 + TF10	-	R4
Authenticate Actors - (1010)						
258	Empty Password in Configuration File	-	1	TF5	-	R5
259	Use of Hard-coded Password	-	-	R2	-	R2
262	Not Using Password Aging	1	1	TF11	-	R5
263	Password Aging with Long Expiration	1	1	TF11	-	R5
287	Improper Authentication	1	1	TF3	1	TF3
288	Authentication Bypass Using an Alternate Path or Channel	1	1	TF3	1	TF3
289	Authentication Bypass by Alternate Name	-	1	TF4	1	TF4
290	Authentication Bypass by Spoofing	1	1	TF4	1	TF4
291	Reliance on IP Address for Authentication	-	1	TF4	1	TF4
293	Using Referer Field for Authentication	-	1	TF4	1	TF4
294	Authentication Bypass by Capture-replay	1	1	TF4	1	TF4
301	Reflection Attack in an Authentication Protocol	-	-	R2	-	R2
302	Authentication Bypass by Assumed-Immutable Data	-	1	TF4	1	TF4
303	Incorrect Implementation of Authentication Algorithm	1	1	TF4	1	TF4
304	Missing Critical Step in Authentication	1	1	TF4	1	TF4
305	Authentication Bypass by Primary Weakness	1	1	TF5	1	TF5
306	Missing Authentication for Critical Function	1	1	TF3	1	TF3
307	Improper Restriction of Excessive Authentication Attempts	1	1	TF5	1	TF4
308	Use of Single-factor Authentication	1	-	R0	-	R0
322	Key Exchange without Entity Authentication	1	1	TF9	1	TF9
521	Weak Password Requirements	1	1	TF5	-	R5
593	Authentication Bypass: OpenSSL CTX Object Modified after SSL Objects are Created	-	-	R2	-	R2
603	Use of Client-Side Authentication	1	-	R2	-	R2
620	Unverified Password Change	1	1	TF5	1	TF5
640	Weak Password Recovery Mechanism for Forgotten Password	1	-	R3	-	R3
798	Use of Hard-coded Credentials	1	-	R2	-	R2
836	Use of Password Hash Instead of Password for Authentication	1	1	TF12	-	R6
916	Use of Password Hash With Insufficient Computational Effort	1	-	R2	-	R2
Authorize Actors - (1011)						
114	Process Control	1	-	R1	-	R1
15	External Control of System or Configuration Setting	1	1	TF6	1	TF6
219	Sensitive Data Under Web Root	-	1	TF3	1	TF3
220	Sensitive Data Under FTP Root	-	1	TF3	1	TF3
266	Incorrect Privilege Assignment	1	1	TF3, TF4	1	TF3, TF4
267	Privilege Defined With Unsafe Actions	1	1	TF3	1	TF3
268	Privilege Chaining	1	1	TF3	1	TF3
269	Improper Privilege Management	1	1	TF3, TF7	1	TF3, TF7
270	Privilege Context Switching Error	1	1	TF7	1	TF7
271	Privilege Dropping / Lowering Errors	1	-	R1	-	R1

Table C.2. Weaknesses related to CWE security design principles view - Authorize Actors (cont.)

CWE ID	Weakness	Gen.	MCP		MST	
			App.	Conditions	App.	Conditions
Authorize Actors - (1011) - (cont.)						
272	Least Privilege Violation	1	-	R1	-	R1
273	Improper Check for Dropped Privileges	1	-	R1	-	R1
274	Improper Handling of Insufficient Privileges	1	-	R1	-	R1
276	Incorrect Default Permissions	1	1	TF3	1	TF3
277	Insecure Inherited Permissions	-	1	TF3	1	TF3
279	Incorrect Execution-Assigned Permissions	-	-	R3	-	R3
280	Improper Handling of Insufficient Permissions or Privileges	1	1	TF3	-	R5
281	Improper Preservation of Permissions	1	-	R1	-	R1
282	Improper Ownership Management	1	-	R1	-	R1
283	Unverified Ownership	1	-	R1	-	R1
284	Improper Access Control	1	1	TF3, TF7	1	TF3, TF7
285	Improper Authorization	1	1	TF3	1	TF3
286	Incorrect User Management	1	1	TF3	1	TF3
300	Channel Accessible by Non-Endpoint ('Man-in-the-Middle')	1	1	TF9	-	R5
341	Predictable from Observable State	1	1	TF8	1	TF8
359	Exposure of Private Information ('Privacy Violation')	1	1	TF2	-	R4
403	Exposure of File Descriptor to Unintended Control Sphere ('File Descriptor Leak')	1	-	R1	-	R1
419	Unprotected Primary Channel	1	1	TF3	1	TF3
420	Unprotected Alternate Channel	1	1	TF3	-	R3
425	Direct Request ('Forced Browsing')	1	1	TF3	1	TF3
426	Untrusted Search Path	1	-	R1	-	R1
434	Unrestricted Upload of File with Dangerous Type	1	1	TF4	1	TF4
527	Exposure of CVS Repository to an Unauthorized Control Sphere	-	-	R1	-	R1
528	Exposure of Core Dump File to an Unauthorized Control Sphere	-	1	TF4	1	TF4
529	Exposure of Access Control List Files to an Unauthorized Control Sphere	-	1	TF3, TF4	1	TF3, TF4
530	Exposure of Backup File to an Unauthorized Control Sphere	-	1	TF3	1	TF3
538	File and Directory Information Exposure	1	1	TF1 + TF2	-	R4
551	Incorrect Behavior Order: Authorization Before Parsing and Canonicalization	1	1	TF3	1	TF3
552	Files or Directories Accessible to External Parties	1	1	TF3	1	TF3
566	Authorization Bypass Through User-Controlled SQL Primary Key	-	-	R1	-	R1
639	Authorization Bypass Through User-Controlled Key	1	1	TF4	1	TF4
642	External Control of Critical State Data	1	1	TF3, TF4, TF6	1	TF3, TF4, TF6
647	Use of Non-Canonical URL Paths for Authorization Decisions	-	1	TF3	1	TF3
653	Insufficient Compartmentalization	1	-	R2	-	R2
656	Reliance on Security Through Obscurity	1	-	R2	-	R2
668	Exposure of Resource to Wrong Sphere	1	1	TF3, TF4, TF6	1	TF3, TF4, TF6
669	Incorrect Resource Transfer Between Spheres	1	1	TF4	1	TF4
671	Lack of Administrator Control over Security	1	-	R2	-	R2
673	External Influence of Sphere Definition	1	-	R2	-	R2
708	Incorrect Ownership Assignment	1	-	R1	-	R1
732	Incorrect Permission Assignment for Critical Resource	1	1	TF3	1	TF3
770	Allocation of Resources Without Limits or Throttling	1	1	TF4	1	TF4
782	Exposed IOCTL with Insufficient Access Control	-	-	R1	-	R1
827	Improper Control of Document Type Definition	-	-	R2	-	R2
862	Missing Authorization	1	1	TF3	1	TF3
863	Incorrect Authorization	1	1	TF3	1	TF3
921	Storage of Sensitive Data in a Mechanism without Access Control	1	1	TF2	-	R1
923	Improper Restriction of Communication Channel to Intended Endpoints	1	1	TF3, TF4	1	TF3, TF4
939	Improper Authorization in Handler for Custom URL Scheme	1	-	R1	-	R1
942	Overly Permissive Cross-domain Whitelist	-	-	R2	-	R2

Table C.3. Weaknesses related to CWE security design principles view - Cross Cutting & Encrypt Data

CWE ID	Weakness	Gen.	MCP		MST	
			App.	Conditions	App.	Conditions
Cross Cutting - (1012)						
208	Information Exposure Through Timing Discrepancy	1	-	R3	-	R3
392	Missing Report of Error Condition	1	1	TF3	1	TF3
460	Improper Cleanup on Thrown Exception	1	-	R2	-	R2
544	Missing Standardized Error Handling Mechanism	1	-	R2	-	R2
602	Client-Side Enforcement of Server-Side Security	1	1	TF4	1	I4
703	Improper Check or Handling of Exceptional Conditions	1	1	TF3	1	TF3
754	Improper Check for Unusual or Exceptional Conditions	1	-	R2	-	R2
784	Reliance on Cookies without Validation and Integrity Checking in a Security Decision	-	1	TF4	1	TF4
807	Reliance on Untrusted Inputs in a Security Decision	1	1	TF4	1	TF4
Encrypt Data - (1013)						
256	Unprotected Storage of Credentials	1	-	R2	-	R2
257	Storing Passwords in a Recoverable Format	1	-	R2	-	R2
260	Password in Configuration File	1	-	R2	-	R2
261	Weak Cryptography for Passwords	1	-	R2	-	R2
311	Missing Encryption of Sensitive Data	1	1	TF1, TF9	1	TF9
312	Cleartext Storage of Sensitive Information	1	1	TF1, TF9	1	TF9
313	Cleartext Storage in a File or on Disk	-	-	R2	-	R2
314	Cleartext Storage in the Registry	-	-	R0	-	R0
315	Cleartext Storage of Sensitive Information in a Cookie	-	1	TF9	-	R5
316	Cleartext Storage of Sensitive Information in Memory	-	-	R0	-	R0 + R5
317	Cleartext Storage of Sensitive Information in GUI	-	-	R2	-	R2 + R5
318	Cleartext Storage of Sensitive Information in Executable	-	-	R2	-	R2
319	Cleartext Transmission of Sensitive Information	1	1	TF9	-	R5
321	Use of Hard-coded Cryptographic Key	-	-	R2	-	R2
323	Reusing a Nonce, Key Pair in Encryption	-	-	R2	-	R2
324	Use of a Key Past its Expiration Date	1	-	R2	-	R2
325	Missing Required Cryptographic Step	1	-	R2	-	R2
326	Inadequate Encryption Strength	1	-	R2	-	R2
327	Use of a Broken or Risky Cryptographic Algorithm	1	-	R2	-	R2
328	Reversible One-Way Hash	1	-	R2	-	R2
330	Use of Insufficiently Random Values	1	-	R2, R3	-	R2, R3
331	Insufficient Entropy	1	-	R0	-	R0
332	Insufficient Entropy in PRNG	-	-	R0	-	R0
333	Improper Handling of Insufficient Entropy in TRNG	-	-	R0	-	R0
334	Small Space of Random Values	1	-	R2, R0	-	R2, R0
335	Incorrect Usage of Seeds in Pseudo-Random Number Generator (PRNG)	1	-	R0	-	R0
336	Same Seed in Pseudo-Random Number Generator (PRNG)	-	-	R2, R0	-	R2, R0
337	Predictable Seed in Pseudo-Random Number Generator (PRNG)	-	-	R0	-	R0
338	Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)	1	-	R0	-	R0
339	Small Seed Space in PRNG	-	-	R2, R0	-	R2, R0
347	Improper Verification of Cryptographic Signature	1	-	R0	-	R0
522	Insufficiently Protected Credentials	1	1	TF3	1	TF3
523	Unprotected Transport of Credentials	1	1	TF9	1	TF9
757	Selection of Less-Secure Algorithm During Negotiation ('Algorithm Downgrade')	1	1	TF4	1	TF4
759	Use of a One-Way Hash without a Salt	-	-	R2	-	R2
760	Use of a One-Way Hash with a Predictable Salt	-	-	R2	-	R2
780	Use of RSA Algorithm without OAEP	-	-	R2	-	R2
922	Insecure Storage of Sensitive Information	1	-	R2	-	R2

Table C.4. Weaknesses related to CWE security design principles view - Identify Actors, Limit Access, Limit Exposure, Lock Computer, Manage User Sessions, and Verify Message Integrity

CWE ID	Weakness	Gen.	MCP		MST	
			App.	Conditions	App.	Conditions
Identify Actors - (1014)						
295	Improper Certificate Validation	1	1	TF14	1	TF14
296	Improper Following of a Certificate's Chain of Trust	1	1	TF14	1	TF14
297	Improper Validation of Certificate with Host Mismatch	-	1	TF14	1	TF14
298	Improper Validation of Certificate Expiration	-	1	TF14	1	TF14
299	Improper Check for Certificate Revocation	1	1	TF14	1	TF14
345	Insufficient Verification of Data Authenticity	1	1	TF13	-	R5
346	Origin Validation Error	1	1	TF3, TF4	1	TF3, TF4
370	Missing Check for Certificate Revocation after Initial Check	-	1	TF14	1	TF14
441	Unintended Proxy or Intermediary ('Confused Deputy')	1	1	TF3, TF4	1	TF3, TF4
599	Missing Validation of OpenSSL Certificate	-	1	TF14	1	TF14
940	Improper Verification of Source of a Communication Channel	1	-	R1	-	R1
941	Incorrectly Specified Destination in a Communication Channel	1	-	R1	-	R1
Limit Access - (1015)						
201	Information Exposure Through Sent Data	1	1	TF2 + TF9	-	R5
209	Information Exposure Through an Error Message	1	-	R3	-	R3
212	Improper Cross-boundary Removal of Sensitive Data	1	1	TF9	1	TF9
243	Creation of chroot Jail Without Changing Working Directory	-	-	R1	-	R1
250	Execution with Unnecessary Privileges	1	-	R3	-	R3
610	Externally Controlled Reference to a Resource in Another Sphere	1	1	TF3, TF6, TF13	1	TF3, TF6, TF13
611	Improper Restriction of XML External Entity Reference	1	1	TF13	-	R6
73	External Control of File Name or Path	1	1	TF3	1	TF3
Limit Exposure - (1016)						
210	Information Exposure Through Self-generated Error Message	1	-	R2, R3	-	R2, R3
211	Information Exposure Through Externally-Generated Error Message	1	-	R2, R3	-	R2, R3
214	Information Exposure Through Process Environment	1	-	R3	-	R3
550	Information Exposure Through Server Error Message	-	-	R3	-	R3
829	Inclusion of Functionality from Untrusted Control Sphere	1	-	R2, R3	-	R2, R3
830	Inclusion of Web Functionality from an Untrusted Source	-	-	R2, R3	-	R2, R3
Lock Computer - (1017)						
645	Overly Restrictive Account Lockout Mechanism	1	1	TF5	-	R5
Manage User Sessions - (1018)						
384	Session Fixation	-	1	TF13	1	TF13
488	Exposure of Data Element to Wrong Session	1	1	TF13	1	TF13
579	J2EE Bad Practices: Non-serializable Object Stored in Session	-	-	R1	-	R1
6	J2EE Misconfiguration: Insufficient Session-ID Length	-	-	R3	-	R3
613	Insufficient Session Expiration	1	1	TF4 + TF11	1	TF4 + TF11
841	Improper Enforcement of Behavioral Workflow	1	1	TF3	1	TF3
Verify Message Integrity - (1020)						
353	Missing Support for Integrity Check	1	-	R2	-	R2
354	Improper Validation of Integrity Check Value	1	-	R2	-	R2
390	Detection of Error Condition Without Action	1	-	R2	-	R2
391	Unchecked Error Condition	1	-	R2	-	R2
494	Download of Code Without Integrity Check	1	-	R2	-	R2
565	Reliance on Cookies without Validation and Integrity Checking	1	1	I4	1	I4
649	Reliance on Obfuscation or Encryption of Security-Relevant Inputs without Integrity Checking	1	-	R2	-	R2
707	Improper Enforcement of Message or Data Structure	1	1	TF3, TF4	1	TF3, TF4
755	Improper Handling of Exceptional Conditions	1	1	TF3	-	R5
924	Improper Enforcement of Message Integrity During Transmission in a Communication Channel	1	-	R2	-	R2

Table C.5. Weaknesses related to CWE security design principles view - Validate Inputs

CWE ID	Weakness	Gen.	MCP		MST	
			App.	Conditions	App.	Conditions
Validate Inputs - (1019)						
138	Improper Neutralization of Special Elements	1	1	TF3	1	TF3
150	Improper Neutralization of Escape, Meta, or Control Sequences	-	1	TF1	-	R6
20	Improper Input Validation	1	1	TF3, TF4	1	TF3, TF4
349	Acceptance of Extraneous Untrusted Data With Trusted Data	1	-	R1	-	R1
352	Cross-Site Request Forgery (CSRF)	-	1	TF13	-	R6
472	External Control of Assumed-Immutable Web Parameter	1	1	TF4	1	TF4
473	PHP External Variable Modification	-	1	TF4	1	TF4
502	Deserialization of Untrusted Data	1	-	R2	-	R2
59	Improper Link Resolution Before File Access ('Link Following')	1	1	TF4	-	R6
601	URL Redirection to Untrusted Site ('Open Redirect')	1	1	TF3 + TF9, TF4	1	TF4
641	Improper Restriction of Names for Files and Other Resources	1	-	R2	-	R2
643	Improper Neutralization of Data within XPath Expressions ('XPath Injection')	1	1	TF4	1	TF4
652	Improper Neutralization of Data within XQuery Expressions ('XQuery Injection')	1	1	TF4	1	TF4
74	Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')	1	1	TF3	1	TF3
75	Failure to Sanitize Special Elements into a Different Plane (Special Element Injection)	1	1	TF3	1	TF3
76	Improper Neutralization of Equivalent Special Elements	1	1	TF3	1	TF3
77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	1	1	TF3	1	TF3
78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	1	1	TF3	1	TF3
79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	1	1	TF13	-	R6
790	Improper Filtering of Special Elements	1	1	TF3	1	TF3
791	Incomplete Filtering of Special Elements	1	1	TF3	1	TF3
792	Incomplete Filtering of One or More Instances of Special Elements	-	1	TF3	1	TF3
793	Only Filtering One Instance of a Special Element	-	1	TF3	1	TF3
794	Incomplete Filtering of Multiple Instances of Special Elements	-	1	TF3	1	TF3
795	Only Filtering Special Elements at a Specified Location	1	1	TF3, TF4	1	TF3, TF4
796	Only Filtering Special Elements Relative to a Marker	-	1	TF3, TF4	1	TF3, TF4
797	Only Filtering Special Elements at an Absolute Position	-	1	TF3, TF4	1	TF3, TF4
88	Improper Neutralization of Argument Delimiters in a Command ('Argument Injection')	1	1	TF4	1	TF4
89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	1	1	TF3, TF4	1	TF3, TF4
90	Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')	1	1	TF4	1	TF4
91	XML Injection (aka Blind XPath Injection)	1	1	TF4	1	TF4
93	Improper Neutralization of CRLF Sequences ('CRLF Injection')	1	1	TF3, TF4	1	TF4
94	Improper Control of Generation of Code ('Code Injection')	1	1	TF4	-	R6
943	Improper Neutralization of Special Elements in Data Query Logic	1	1	TF3, TF4	1	TF3, TF4
95	Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection')	-	1	TF4	-	R6
96	Improper Neutralization of Directives in Statically Saved Code ('Static Code Injection')	1	-	R2	-	R2
97	Improper Neutralization of Server-Side Includes (SSI) Within a Web Page	-	-	R2	-	R2
98	Improper Control of Filename for Include/Require Statement in PHP Program ('PHP Remote File Inclusion')	-	-	R2	-	R2
99	Improper Control of Resource Identifiers ('Resource Injection')	1	1	TF4	-	R6

Table C.6. Weaknesses related to OWASP Top 10 Security Risks - Categories A1-A3

CWE ID	**Weakness**	**Gen.**	**MCP**		**MST**	
			App.	**Conditions**	**App.**	**Conditions**
OWASP Top Ten 2017 Category A1 - Injection - (1027)						
77	Improper Neutralization of Special Elements used in a Command ('Command Injection') - (77)	1	1	TF3	1	TF3
78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') - (78)	1	1	TF3	1	TF3
88	Improper Neutralization of Argument Delimiters in a Command ('Argument Injection') - (88)	1	1	TF4	1	TF4
89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') - (89)	1	1	TF3, TF4	1	TF3, TF4
90	Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection') - (90)	1	1	TF4	1	TF4
91	XML Injection (aka Blind XPath Injection) - (91)	1	1	TF4	1	TF4
564	SQL Injection: Hibernate - (564)	-	-	R3	-	R3
917	Improper Neutralization of Special Elements used in an Expression Language Statement ('Expression Language Injection') - (917)	1	1	TF3	-	R2
943	Improper Neutralization of Special Elements in Data Query Logic - (943)	1	1	TF3, TF4	1	TF3, TF4
OWASP Top Ten 2017 Category A2 - Broken Authentication - (1028)						
287	Improper Authentication - (287)	1	1	TF3	1	TF3
256	Unprotected Storage of Credentials - (256)	1	-	R2	-	R2
308	Use of Single-factor Authentication - (308)	1	-	R0	-	R0
384	Session Fixation - (384)	-	1	TF13	1	TF13
522	Insufficiently Protected Credentials - (522)	1	1	TF3	1	TF3
523	Unprotected Transport of Credentials - (523)	1	1	TF9	1	TF9
613	Insufficient Session Expiration - (613)	1	1	TF4 + TF11	1	TF4 + TF11
620	Unverified Password Change - (620)	1	1	TF5	1	TF5
640	Weak Password Recovery Mechanism for Forgotten Password - (640)	1	-	R3	-	R3
OWASP Top Ten 2017 Category A3 - Sensitive Data Exposure - (1029)						
220	Storage of File With Sensitive Data Under FTP Root - (220)	-	1	TF3	1	TF3
295	Improper Certificate Validation - (295)	1	1	TF14	1	TF14
311	Missing Encryption of Sensitive Data - (311)	1	1	TF1, TF9	1	TF9
312	Cleartext Storage of Sensitive Information - (312)	1	1	TF1, TF9	1	TF9
319	Cleartext Transmission of Sensitive Information - (319)	1	1	TF9	-	R5
320	Key Management Errors - (320)	1	-	R2	-	R2
325	Missing Required Cryptographic Step - (325)	1	-	R2	-	R2
326	Inadequate Encryption Strength - (326)	1	-	R2	-	R2
327	Use of a Broken or Risky Cryptographic Algorithm - (327)	1	-	R2	-	R2
328	Reversible One-Way Hash - (328)	1	-	R2	-	R2
359	Exposure of Private Personal Information to an Unauthorized Actor - (359)	1	1	TF2	-	R4

Table C.7. Weaknesses related to OWASP Top 10 Security Risks - Categories A4-A10

CWE ID	Weakness	Gen.	MCP		MST	
			App.	Conditions	App.	Conditions
OWASP Top Ten 2017 Category A4 - XML External Entities (XXE) - (1030)						
611	Improper Restriction of XML External Entity Reference - (611)	1	1	TF13	-	R6
776	Improper Restriction of Recursive Entity References in DTDs ('XML Entity Expansion') - (776)	1	-	R2	-	R2
OWASP Top Ten 2017 Category A5 - Broken Access Control - (1031)						
22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal') - (22)	1	1	TF3	1	TF3
284	Improper Access Control - (284)	1	1	TF3, TF7	1	TF3, TF7
285	Improper Authorization - (285)	1	1	TF3	1	TF3
425	Direct Request ('Forced Browsing') - (425)	1	1	TF3	1	TF3
639	Authorization Bypass Through User-Controlled Key - (639)	1	1	TF4	1	TF4
OWASP Top Ten 2017 Category A6 - Security Misconfiguration - (1032)						
16	Configuration - (16)	1	1	TF3, TF4	1	TF3, TF4
209	Generation of Error Message Containing Sensitive Information - (209)	1	-	R3	-	R3
548	Exposure of Information Through Directory Listing - (548)	-	1	TF3	1	TF3
OWASP Top Ten 2017 Category A7 - Cross-Site Scripting (XSS) - (1033)						
79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') - (79)	1	1	TF13	-	R6
OWASP Top Ten 2017 Category A8 - Insecure Deserialization - (1034)						
502	Deserialization of Untrusted Data - (502)	1	-	R2	-	R2
OWASP Top Ten 2017 Category A9 - Using Components with Known Vulnerabilities - (1035)						
OWASP Top Ten 2017 Category A10 - Insufficient Logging & Monitoring - (1036)						
223	Omission of Security-relevant Information - (223)	1	-	R2	-	R2
778	Insufficient Logging - (778)	1	1	TF1 + TF10	-	R4

Table C.8. CWE Top 25 Most Dangerous Software Errors View

CWE ID	Weakness	Gen.	MCP		MST	
			App.	Conditions	App.	Conditions
119	Improper Restriction of Operations within the Bounds of a Memory Buffer	1	-	R2	-	R2
79	Cross-site Scripting	1	1	TF13	-	R6
20	Improper Input Validation	1	1	TF3, TF4	1	TF3, TF4
200	Information Exposure	1	1	TF2 + TF9, TF2	-	R3
125	Out-of-bounds Read	1	-	R2	-	R2
89	SQL Injection	1	1	TF3, TF4	1	TF3, TF4
416	Use After Free	-		R2	-	R2
190	Integer Overflow or Wraparound	1	1	TF4	-	R3
352	Cross-Site Request Forgery (CSRF)	-		TF13	-	R6
22	Path Traversal	1	1	TF3	1	TF3
78	OS Command Injection	1	1	TF3	1	TF3
787	Out-of-bounds Write	1	-	R2	-	R2
287	Improper Authentication	1	1	TF3	1	TF3
476	NULL Pointer Dereference	1	-	R2	-	R2
732	Incorrect Permission Assignment for Critical Resource	1	1	TF3	1	TF3
434	Unrestricted Upload of File with Dangerous Type	1	1	TF4	1	TF4
611	Improper Restriction of XML External Entity Reference	1	1	TF13	-	R6
94	Code Injection	1	1	TF4	-	R6
798	Use of Hard-coded Credentials	1	-	R2	-	R2
400	Uncontrolled Resource Consumption	1	-	R2	-	R2
772	Missing Release of Resource after Effective Lifetime	1	-	R3	-	R3
426	Untrusted Search Path	1	-	R1	-	R1
502	Deserialization of Untrusted Data	1	-	R2	-	R2
269	Improper Privilege Management	1	1	TF3, I10	1	TF3, I10
295	Improper Certificate Validation	1	1	TF14	1	TF14