Scalable Metamorphic Testing Approach for Web System Security

by

Nazanin Bayati Chaleshtari

Thesis submitted to the University of Ottawa
in partial fulfillment of the requirements
for the Doctorate of Philosophy degree in
Computer Engineering

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Abstract

Security testing aims at verifying that the software meets its security properties. In modern Web systems, however, this often entails the verification of the outputs generated when exercising the system with a very large set of inputs. Full automation is thus required to lower costs and increase the effectiveness of security testing. Unfortunately, to achieve such automation, in addition to strategies for automatically deriving test inputs, we need to address the oracle problem, which refers to the challenge, given an input for a system, of distinguishing correct from incorrect behaviour (e.g., the response to be received after a specific HTTP GET request). Moreover, another important objective in security testing is to discover vulnerabilities within a reasonable timeframe.

This thesis addresses these challenges by first introducing a metamorphic testing approach (MST-wi) that integrates test input generation strategies inspired by mutational fuzzing and alleviates the oracle problem in security testing. It enables engineers to specify metamorphic relations (MRs) that capture many security properties of Web systems. To facilitate the specification of such MRs, we provide a domain-specific language supported by an Eclipse editor. MST-wi automatically collects the input data and transforms the MRs into executable Java code to automatically perform security testing. It automatically tests Web systems to detect vulnerabilities based on the MRs and collected data.

However, metamorphic relations are typically executed on a large set of inputs, which is time-consuming and thus makes metamorphic testing impractical for large systems. We tackle this challenge by proposing an Automated Input set Minimization (AIM) approach that selects inputs to reduce testing costs while preserving vulnerability detection capabilities. AIM includes a clustering-based, black-box approach to identify similar inputs based on their security properties. It also relies on a novel genetic algorithm to efficiently select diverse inputs while minimizing their total cost. Further, it contains a problem-reduction component to reduce the search space and speed up the minimization process.

Acknowledgements

This work was supported by the H2020 COSMOS European project, grant agreement No. 957254, NSERC of Canada under the Discovery and CRC programs, and the Science Foundation Ireland grant 13/RC/2094-2. It is part of a collaborative research program between the University of Ottawa's Nanda laboratory and the SnT center at the University of Luxembourg.

I would like to express my deepest gratitude to my supervisor, Professor Lionel Briand, for his invaluable guidance, support, and encouragement throughout my research journey. His mentorship has been instrumental in my academic and professional development.

My sincere thanks go Fabrizio Pastore, for his collaboration, insights, and constructive feedback, which have significantly enriched my work. Also, I am deeply grateful to Yoann Marquer for his support throughout my research, especially during challenging times.

I am also deeply thankful to my committee members for their insightful feedback and for their valuable feedback, which has greatly enhanced the quality of my research.

Lastly, I would like to thank my family for their unwavering support and encouragement, which have been a constant source of strength and motivation.

Dedication

To my parents.

Glossary of Acronyms

AIM Automated Input set Minimizer

ART Adaptive Random Testing

CWE Common Weakness Enumeration

DSL Domain-Specific Language

GA Genetic Algorithm

IDE Integrated Development Environment

MOSA Many Objective Sorting Algorithm

MST Metamorphic Security Testing

MST-wi Metamorphic Security Testing for Web-interactions

MT Metamorphic Testing

NSGA-III Non-Dominated Sorting Genetic Algorithm III

OWASP Open Web Application Security Project

RQ Research Question

RT Random Testing

SOTA State Of The Art

VDR Vulnerability Detection Rate

XML eXtensible Markup Language

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Web systems (i.e., software systems providing services through Web pages consisting of HTML, Javascript, and CSS) are one of the main means to deliver online services, from e-commerce to online banking. These systems are business-critical, manage critical assets (e.g., card transactions), and often store sensitive information (e.g., customer data). Therefore, it is essential to have an automated approach that helps discover security vulnerabilities (i.e., faults preventing the system from fulfilling its security requirements) in Web systems within a practical amount of time [84, 91, 129–131]. In short, this is the objective of security testing.

Unfortunately, it is hard to discover vulnerabilities in Web systems during testing. Indeed, Web systems typically consist of several input interfaces (i.e., Web pages provided through URL requests). Each interface handles a large set of inputs (e.g., Web forms, cookies, URL parameters) that might be configured differently according to user roles. Considering that vulnerabilities might result from specific combinations of user roles, URL requests, and parameters, it is necessary to exercise the system with a large set of inputs, including inputs crafted to harm the system. Therefore, strategies to automatically generate security test inputs are necessary.

However, automatically generating inputs is insufficient; indeed, an automated *test oracle* (i.e., a mechanism for determining whether a test case has passed or failed) is needed. Security testing is known to suffer from the oracle problem [43, 191, 218]. It is indeed generally infeasible to automatically determine the expected output for a given input or even manually specify expected outputs for a large number of test inputs. For instance, a security test case for a bypass authorization schema vulnerability should verify, for every user role, whether it is possible to access resources that should be available only to a user having a different role [146]. We can discover this vulnerability by verifying access to various resources with different privileges and roles; however, to determine the test outcome, we need a mapping between roles and resources, which is not always available because of the complexity of the operations provided by modern Web systems. Recent

incidents involving corporate Web sites, such as Facebook, demonstrate that it is difficult to verify, at testing time, large sets of input sequences, including the ones that trigger vulnerabilities [71, 200, 235, 241].

Although several security testing approaches exist in the literature, they do not address the oracle problem and assume the availability of an implicit test oracle [43]. Furthermore, most of them focus on a particular vulnerability type (e.g., buffer overflows [93, 179]) and only uncover vulnerabilities that prevent a system from providing outputs (e.g., system crashes because of buffer overflows).

Metamorphic Testing (MT) is a testing technique that has shown, in some contexts, to be very effective in alleviating the oracle problem [63, 124]. 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 to specify the system's input-output behavior [210]. Considerable research has been devoted to developing MT approaches for application domains such as computer graphics (e.g., [89, 105, 115, 140]), Web services (e.g., [59, 222, 257]), and embedded systems (e.g., [58, 100, 114, 231]). Unfortunately, only a few works target security aspects [62]; also, their applicability is limited to the functional testing of security components (e.g., code obfuscators [62]) or the verification of specific security bugs (e.g., heartbleed [224]). They do not support the specification of general security properties by using MRs and, although MT is automatable, few MT approaches provide proper tool support [210].

Furthermore, the practicality and scalability of metamorphic testing naturally depends on the number of inputs to be processed. One might consider parallelizing the executions; however, such a solution may not fit all development contexts (e.g., not all companies have the infrastructure to support parallel execution of the software under test and its test cases). Further, even when parallelization is possible, a reduction of the test execution time may provide tangible benefits, including earlier vulnerability detection. In general, what is required is an approach to minimize the number of inputs to be used during testing.

1.2 Thesis Objectives

The long-term vision of this research is to provide a scalable approach that both automatically generates test inputs for the security testing of Web systems and addresses the test oracle and minimization problems.

To achieve this vision, we further refine it into two main objectives:

TO_1 (Metamorphic Security Testing): Use MT to both automatically generate test inputs and address the test oracle problem in security testing.

TO_2 (Input Set Minimization): Reduce the cost of metamorphic security testing by minimizing a set of test inputs, while preserving the capability of MRs to detect security vulnerabilities.

1.3 Contributions

The research contributions of this thesis include:

1. Metamorphic Security Testing for Web-interactions (*MST-wi*), a technique that supports engineers in specifying MRs to capture security properties of Web systems and automatically detect vulnerabilities (i.e., violations of security properties) based on those relations. *MST-wi* is built on top of the following novel solutions:

   - Security Metamorphic Relation Language (SMRL), a Domain-Specific Language (DSL) for specifying MRs for software security testing. SMRL is supported by a custom editor, implemented as a plug-in for the Eclipse IDE [1], which facilitates the specification of MRs.
   - A catalog of 76 system-agnostic MRs targeting 102 known security vulnerability types in Web systems.
   - A data collection framework that automatically collects the data required to perform MT.

2. A testing framework for *MST-wi* that automatically performs security testing based on the MRs and the collected data.

3. A comprehensive empirical evaluation to assess the effectiveness and efficiency of *MST-wi* in discovering vulnerabilities within large-scale web systems.

4. A novel approach, named AIM, to minimize input sets for metamorphic testing while retaining the input sets' ability to detect vulnerabilities. Note that many of the steps in AIM are not specific to Web systems, though others would need to be tailored to other domains (e.g., desktop applications, embedded systems).

   This approach includes the following novel components:

   - An extension of the *MST-wi* framework to retrieve output data and extract cost information about MRs without requiring their execution.
   - A black-box approach leveraging clustering algorithms to partition the initial input set based on security-related characteristics in order to keep a small number of representative inputs.
   - MOCCO (Many-Objective Coverage and Cost Optimizer), a novel genetic algorithm specifically designed to fit our problem and which is able to efficiently select diverse inputs while minimizing their total cost.
   - IMPRO (Inputset Minimization Problem Reduction Operator), an approach to reduce the search space to its minimal extent, and then divide it in smaller, easier to minimize, independent parts.

5. A prototype framework for AIM that automates the process of input set minimization for Web systems.

6. An extensive empirical evaluation (about 800 hours of computation) aimed at assessing the effectiveness of AIM in terms of vulnerability detection and performance, considering 18 different AIM configurations and 5 search algorithms (including MOCCO) for security testing, on the Jenkins and Joomla systems, which are the most used Web-based frameworks for development automation and context management.

1.4 Publications

In addition to this thesis, the research results have been communicated to the public via the following publications in the top software engineering venue:

- **N. Bayati Chaleshtari**, F. Pastore, A. Goknil, and L. Briand, "Metamorphic Testing for Web System Security," in IEEE Transactions on Software Engineering, 2023, doi: 10.1109/TSE.2023.3256322.
- **N. Bayati Chaleshtari**, Y. Marquer, F. Pastore, and L. Briand, "AIM: Automated Input Set Minimization for Metamorphic Security Testing," in IEEE Transactions on Software Engineering, 2024, doi: 10.1109/TSE.2024.3488525.

The detailed contributions of each publication are listed in Section 1.3, where contributions 1-3 correspond to publication 1; contributions 4-6 correspond to publication 2.

1.5 Thesis Outline

The rest of this thesis is structured as follows:

- Chapter 2 addresses TO_1 by providing:
  - Background on metamorphic testing
  - Overview of the proposed metamorphic security testing approach with a description of the core technical solutions
  - A catalog of our defined MRs
  - Security vulnerability types that our proposed approach can detect and testability guidelines
  - A thorough empirical evaluation conducted with two open-source case studies
  - Discussion of threats to validity
  - Review of related works and their shortcomings in addressing the defined problem
  - Conclusion of our work

- Chapter 3 starts addressing TO_2 by providing:
  - Background information necessary to state our problem and detail our approach
  - Detailed problem definition related to minimizing the initial input set while retaining inputs capable of detecting distinct software vulnerabilities
  - Overview of the proposed input set minimization approach with a description of the core technical solutions
  - An empirical evaluation of the proposed technique
  - Discussion of threats to validity
  - Review of related works and comparison with our proposed technique
  - Conclusion of our work
- Finally, Chapter 4 provides:
  - A summary of the key contributions of the thesis
  - Directions for future research
  - Final remarks on the significance of the work

Chapter 2

Metamorphic Security Testing

This chapter addresses TO_1, i.e., utilizing MT to both automatically generate test inputs and address the test oracle problem in security testing of Web systems. The contents of this chapter have been published in the Journal of *Transactions on Software Engineering (TSE)* [47].

2.1 Overview

As discussed in chapter 1, it is challenging to discover vulnerabilities in Web systems during testing, and it is necessary to utilize strategies to automatically generate security test inputs. When a large number of test inputs are needed, even in the presence of automated generation, testing becomes impractical if we lack solutions to automatically derive test oracles. Security testing is known to suffer from the oracle problem. MT is a testing technique that has shown, in some contexts, to be very effective in alleviating the oracle problem [63, 124].

Our goal in this study is to use MT to both automatically generate test inputs for security testing and alleviate the test oracle problem in security testing. We propose Metamorphic Security Testing for Web-interactions (*MST-wi*), a technique that supports engineers in specifying MRs to capture security properties of Web systems and automatically detect vulnerabilities (i.e., violations of security properties) based on those relations.

MST-wi automatically generates test inputs by altering valid inputs as an attacker might do; the strategies adopted to modify valid inputs are encoded in MRs. It automatically collects valid inputs by relying on a Web crawler or reusing the scripts engineers implement for functional testing. Then, for a given vulnerability, we rely on available resources, including its description and its demonstrative examples, to understand attacker's actions. Then, based on the collected information from our inspection, we implement an MR. For example, an MR to spot bypass authorization schema vulnerabilities should ensure that *a Web system returns different responses to two users when the first user (User-A) requests a URL that is provided to her by the GUI (e.g., in HTML links) while the second user (User-B) requests the same URL, which is not provided to her by the GUI.* With

MST-wi, it is possible to define an executable MR that accesses, for User-B, all the URLs that can be reached by the GUI for User-A but not with the GUI for User-B. The MR verifies that the output returned to User-B is different from the one for User-A; otherwise, a vulnerability is reported. If the output for the two users is the same, User-B can access the same page accessed by User-A instead of receiving an error message, which is not the functionality exposed by the user interface.

MST-wi includes the following contributions:

- Security Metamorphic Relation Language (SMRL), a Domain-Specific Language (DSL) for specifying MRs for software security testing. SMRL is supported by a custom editor, implemented as a plug-in for the Eclipse IDE [1], which facilitates the specification of MRs.
- A catalog of system-agnostic MRs targeting 102 security vulnerabilities in Web systems.
- A data collection framework that automatically collects the data required to perform MT.
- A testing framework that automatically performs security testing based on the MRs and the collected data.

Our DSL supports data representation functions and boolean operators to specify security properties in MRs. It also provides a set of utility functions to express data properties that cannot be described with simple boolean or arithmetic operators. *MST-wi* automatically transforms MRs written in our DSL into executable Java code. It extends the Crawljax Web crawler [144] to derive source inputs automatically from the system under test. It provides an MT algorithm integrated into the JUnit framework [4] as part of the testing framework. The algorithm performs MT based on executable MRs in Java and source inputs collected by the data collection framework.

We evaluated our approach through the following analyses:

- The capability of *MST-wi* to automate (including oracles) the security testing activities suggested by OWASP[1]. Our results show that *MST-wi* automates 39% of the security testing activities not automated by other approaches relying on catalogs or implicit oracles (e.g., crashes) and detecting specific vulnerability types.
- The applicability of *MST-wi* to discover common and important security vulnerability types described in the Common Weaknesses Enumeration Repository [26]. Our results show that *MST-wi* enables testing for 101 (45%) vulnerability types due to errors in applying security design principles.

[1]OWASP (Open Web Application Security Project) is one of the best known organizations that focus on software security [27].

- A study of testability guidelines that enable engineers to improve the testability of Web systems with *MST-wi*. We observe that controllability and an adequate test support environment are key for applying *MST-wi*.

- An analysis of the effectiveness of *MST-wi* when applied to discover vulnerabilities in Jenkins, a leading open source automation server [77], and Joomla, a content management system [2]. *MST-wi* has shown to be largely effective; indeed, it automatically detected 85% of the targeted vulnerabilities affecting these two systems. Further, only a negligible fraction of follow-up test inputs leads to false alarms (0.19% max).

- An analysis of the efficiency of *MST-wi* while testing Jenkins and Joomla. Our results show that, for most vulnerability types, *MST-wi* can be applied overnight; further, technical improvements in our framework implementation may enable overnight execution for all the MRs in our catalog.

This research brings together, refines, and significantly extends the ideas from two papers [128, 132]. We largely extend a conference paper published at the 13th IEEE International Conference on Software Testing, Verification and Validation (ICST'20) [132]. Moreover, an earlier version of the tool presented in this chapter was demonstrated at the 42nd International Conference on Software Engineering (ICSE'20) [128]. The following provides a detailed description of the extension:

- We extend the DSL with additional data representation and utility functions.

- We extend the MR catalog with 54 new MRs.

- We present the results of an extensive study on the types of security vulnerabilities that can be addressed by our approach. Precisely, we study the security weaknesses organized in the CWE view for common security architectural tactics [6], the ones belonging to the CWE Top-25 most dangerous software errors [5], and the ones in the OWASP Top-10 Web security risks [11].

- We provide testability guidelines that assist engineers in designing and implementing their software to enable effective test automation with *MST-wi*.

- We provide substantial new empirical evidence to demonstrate the effectiveness and efficiency of *MST-wi* by applying all the MRs in our catalog to Jenkins and Joomla. Our results include the discovery of a new security vulnerability in Jenkins that received a CVE identifier [127].

Chapter Structure. Section 2.2 provides the background information regarding MT. In Section 2.3, we present an overview of the approach. Sections 2.4 to 2.7 describe the core technical solutions. Section 2.8 describes our catalog of MRs. In Section 2.9, we investigate the security vulnerability types that *MST-wi* can detect and the testability guidelines for *MST-wi* to identify these vulnerabilities. Section 2.10 reports on the results of the empirical validation conducted with two open-source case studies. Section 2.11 addresses threats to validity. Section 2.12 discusses the related work. We conclude the study in Section 2.13.

2.2 Background: Metamorphic Testing

In this section, we present the basic concepts of MT. The core of MT is a set of MRs, which are necessary properties of the program under test in relation to multiple inputs and their expected outputs [66]. MRs *resemble the traditional concept of program invariants, which are properties that hold at certain points in programs. However, the key difference is that an invariant has to hold for every possible program execution, whereas a metamorphic relation captures a property of inputs and outputs belonging to different executions* [210].

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 verifying the outputs of different executions against the MR. Below we provide the 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. An MR is a relation concerning a set of inputs $\langle x_1, ..., x_n \rangle$, where $n \geq 2$, and a set of outputs generated with those inputs $\langle f(x_1), ..., f(x_n) \rangle$. MRs are typically expressed as implications.

Definition 2 (Source Input and Follow-up Input). An MR 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 satisfies the properties expressed by the MR.

Follow-up inputs can be obtained by applying *transformation functions* to the source inputs. The use of *transformation functions* in MRs simplifies the identification of follow-up inputs.

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

MT-1 Generate a number of source inputs (usually one) required by the MR.

MT-2 Execute the MR, which implies the following steps (they can be executed in any order, depending on the MR):

- MT-2.1 Execute the function under test with the source input(s).
- MT-2.2 Derive follow-up input(s) based on the MR.
- MT-2.3 Execute the function under test with the follow-up input(s).

MT-3 Check whether the results violate the MR. If the MR is violated, then the function under test is faulty.

MT-4 Restart from (MT-1), up to a predefined number of iterations.

As an example, let us consider an algorithm f that computes the shortest path for an undirected graph G as described in [47,132]. For any two nodes a and b in graph G, it may not be feasible to generate all possible paths from a to b and 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 remains the same if nodes a and b are swapped. By 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 by the relation. If the relation is violated, f is faulty.

A source input in the example consists of a (random) graph G to be generated and two vertices a and b in G to be randomly selected. Follow-up inputs can be generated by a *transformation function* that swaps the last two arguments of the source input. We call this function *swapLastArguments* and apply it to (G, a, b).

Based on the above, our MR can thus be defined as follows:
$x_1 = (G, a, b) \wedge x_2 = \mathit{swapLastArguments}(x_1) \rightarrow |f(x_1)| = |f(x_2)|$

We first generate a (random) graph G and randomly select two vertices a and b in G for the source input (see MT-1). We then execute the MR (MT-2). We execute the shortest path function twice: first with (G, a, b) (MT-2.1) and then with (G, b, a) (MT-2.3). We derive the follow-up input by applying function *swapLastArguments* to (G, a, b) (MT-2.2). Finally, in MT-3, we verify that the absolute value of the two retrieved outputs is the same; otherwise, we report a failure.

2.3 Overview of the Approach

The process in Fig. 2.1 presents an overview of *MST-wi*.

In Step 1, the engineer selects, from a catalog of predefined MRs, the relations for the system under test. In general, we expect the engineer to choose the whole set of MRs in our catalog. In addition, the engineer can also specify new relations by using our DSL. Step 1 is manual.

In Step 2, *MST-wi* automatically transforms the selected MRs into executable Java code.

In Step 3, the engineer executes a Web crawler to automatically collect information about the Web system under test (hereafter, SUT). The crawler reveals the SUT structure (e.g., URLs that can be visited by an anonymous user) and the actions that trigger the generation of new content on a page (e.g., clicking on an anchor or a textual label). The collected data is used to derive source inputs for MT. The engineer can process manually implemented test scripts, if available, to obtain additional information. Step 3 does not depend on other steps.

In Step 4, *MST-wi* automatically loads source inputs and generates follow-up inputs as described by the relations. After executing the source and follow-up inputs, the outputs are checked according to the MRs.

Our DSL and the data collection framework can be extended to support new language constructs and data collection methods. The MT framework can also be improved to deal

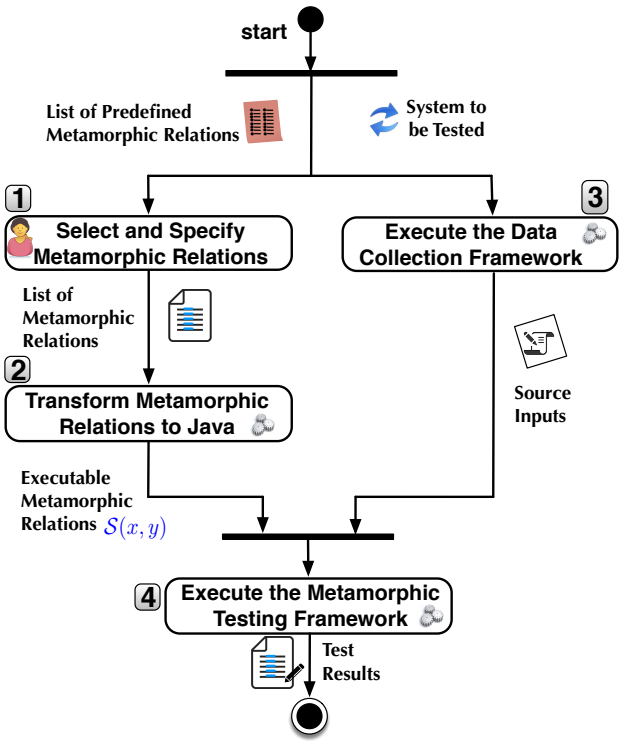

Figure 2.1: Overview of the approach [47, 132].

with input interfaces not supported yet (e.g., Silverlight plug-ins [147]) and to load data collected by new data collection methods [47, 132].

Sections 2.4 to 2.7 explain the details of each step in Fig. 2.1, with a focus on how we achieved our automation objectives.

2.4 Step 1: Select and Specify MRs

Step 1 in Fig. 2.1 concerns selecting and specifying MRs. To enable specifying new MRs, we provide a DSL called Security Metamorphic Relation Language (SMRL).

The most common usage scenario for *MST-wi* is the selection of MRs for the SUT from our MRs catalog (see Section 2.8) without specifying additional ones. Since our catalog covers generic vulnerabilities that may affect any Web system, we expect all the MRs to be chosen in most cases. Further, we name the MRs based on the CWE IDs they aim to detect. For example, an engineer targeting XSS injection, with CWE ID 79, can select and run MR_79.

In our implementation, the catalog of MRs is released as an Eclipse project folder, including our MRs. The project can be copied and reconfigured for every SUT. The selection of MRs is performed within the Eclipse environment by defining a JUnit test suite containing the MRs to test (see Section 2.7).

```
CWE_266_267_268_269_285_522_529_862_863_OTG_AUTHZ_002.smrl
1  import static smrl.mr.language.Operations.*
2  import smrl.mr.language.Action;
3
4  package smrl.mr.owasp {
5
6     MR CWE_266_267_268_269_285_522_529_862_863_OTG_AUTHZ_002 {
7     {
8       for ( Action action : Input(1).actions() ){                                    //(1)
9
10        IMPLIES(
11             (!isSupervisorOf(User(), action.user)) &&                               //(2)
12             cannotReachThroughGUI( User(), action.url )&&                           //(3)
13             CREATE( Input(2), changeCredentials(Input(1), User()) )                 //(4)
14          ,
15          OR(
16             isError(Output(Input(1),action.position)),                              //(5)
17             NOT( Output(Input(1),action.position).equals(Output(Input(2),action.position)))
18       )); //end-IMPLIES
19     } //end-for
20    }} //end-MR
21   }//end-package
```

Description of the annotated MR statements: (1) For loop iterates over all actions of the Input. (2) Checks whether the user in User() is not a supervisor of the user performing the current action. (3) Verifies that the user cannot retrieve the URL of the action through the GUI (based on the data collected by the crawler). (4) Defines a follow-up input that matches the source input except that the credentials of User() are used in this case. (5) Verifies that the follow-up input leads to an error page or the output generated by the action containing the URL indicated above leads to two different outputs in the two cases.

Figure 2.2: An MR for the Bypass Authorization Schema vulnerability.

The definition of new MRs requires basic knowledge of Java programming and an understanding of the SMRL DSL, to implement an MR for the targeted weakness.

Since the selection of MRs is straightforward, we focus on the SMRL DSL in the following paragraphs. We introduce the SMRL grammar, the boolean operators, the data representation functions, and the Web-utility functions.

2.4.1 SMRL Grammar

SMRL builds upon Xbase [80], a powerful expression language provided by Xtext [3]. Specifications written in Xbase can be translated into Java programs, which can then be compiled into executable Java bytecode.

We choose Xbase as the foundation because DSLs built on it inherit the syntax of a Java-like expression language, along with essential language infrastructure components such as a parser, linker, compiler, and interpreter [80]. Additionally, Xbase includes advanced features common in modern programming languages, such as operator overloading, lambda expressions, and type inference, which support the adoption of SMRL.

In SMRL we extend the capabilities of Xbase by incorporating (i) functions for data representation, (ii) boolean operators for defining security properties, and (iii) Web-utility functions for expressing properties of data and data transformation. It is possible to extend these functions by implementing additional Java APIs that can be invoked in MRs.

Fig. 2.2 illustrates an MR written using our SMRL editor. This relation verifies whether unauthorized users can access the URLs dedicated to specific users through a direct request. We refer to this example as a running example throughout this chapter.

Table 2.1: Data functions in SMRL.

Category	Data function	Description
Interaction	Input(int i)	Returns the i^{th} input sequence.
	Action(int i)	Returns the i^{th} input action.
	ActionAvailable-WithoutLogin(int i)	Returns the i^{th} input action that can be performed without logging into the system.
	User(int i)	Returns the i^{th} user of the system.
	ParameterValueUsedBy-OtherUsers(Action i, par i)	Returns, for the same action and parameter position, the i^{th} parameter value used by a user different than the one executing the action.
SUT	RandomFilePath(int i)	Returns the i-th file system path. We select paths of files in the Web system sub-folder, ignoring images, and replacing symbolic links (e.g., 'plugins' is mapped to 'plugin' in Jenkins).
	RandomAdmin-FilePath(int i)	Returns the i-th path to configuration file for the SUT.
	Log(int i)	Returns the i-th path to a log file generated by the SUT.
MST-wi	HttpMethod(int i)	Returns the i-th name of an HTTP method (e.g., DELETE).
	SQLInjectionString(int i)	Returns the i-th attack string to be used to perform an SQL injection (e.g., `'or '1' = '1`).
	CodeInjectionString(int i)	Returns the i-th attack string to be used to perform an code injection, e.g., `/%3C?php%20system(%22/bin/ls %20-l%22);?%3E`.
	XSSInjectionString(int i)	Returns the i-th attack string to be used to perform an XSS injection, e.g., `<SCRIPT>alert('XSS');</SCRIPT>`.
	StaticInjectionString(int i)	Returns the i-th attack string to be used to perform an static code injection.
	LDAPInjectionString(int i)	Returns the i-th attack string to be used to perform an LDAP injection.
	XQueryInjection(int i)	Returns the i-th attack string to be used to perform an XQuery injection.
	CommandInjection(int i)	Returns the i-th attack string to be used to perform a command injection.
	CRLFAttackString(int i)	Returns the i-th CRLF attack string, e.g., `');die(2'`.
	WeakPassword(int i)	Returns the i-th weak password to be tested.
	SpecialCharacters(int i)	Returns the i-th special character to be tested.
	FileWithInvalidType(int i)	Returns the path to the i-th file with invalid type to be tested.
	XMLInjectedFile(int i)	Returns the path to the i-th XML file containing an XML injection.
	RandomValue(Type t)	Returns a random value of the given type.
Output	Output(Input i)	Returns the sequence of outputs generated by Input *i*.
	Output(Input i, int n)	Returns the output generated by the n^{th} action of Input *i*.

Note: for each data function in the table, except for the *Output* category, we provide a convenience method that does not include the parameter *int i* and simply returns the first data item for that type; this leads to 46 data functions.

The SMRL grammar build upon the Xbase grammar, which itself is an extension of the Java grammar. Each SMRL specification can incorporate external classes and APIs. In practice, this allows for an arbitrary number of import declarations to specify the APIs utilized within MRs (Line 1 in Fig. 2.2).

A package declaration resembles the stricture of a Java package and can include one or more MRs. Line 4 in Fig. 2.2 declares the package *smrl.mr.owasp*, which is the package for our MRs and it automates the testing activities in the OWASP testing guidelines [146]. Similar to Java, MRs defined in different SMRL specification files can be grouped under the same package.

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

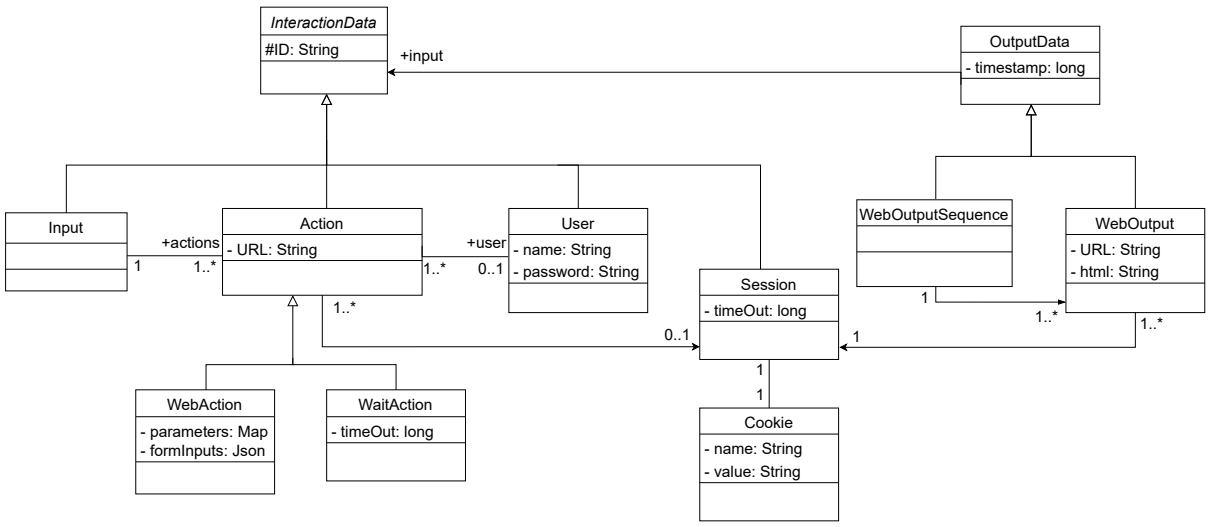

Figure 2.3: Metamorphic data classes in SMRL.

2.4.2 Data Representation Functions

SMRL provides 46 functions to represent different data types used to refer to SUT inputs or outputs in MRs. At a high level, we distinguish between four main categories of data:

- *Interaction data* characterize the interactions with the SUT and is collected by the *MST-wi* Web crawler.
- *SUT data* is specific to the SUT and needs to be specified by the end-user in *MST-wi* configuration files (e.g., the path of log files generated by the SUT).
- *MST-wi data* is provided with the MST framework and does not need to be modified by the end-user (e.g., attack vectors for SQL injection attacks).
- *Output data* is generated by the SUT in response to an input (e.g., Web pages).

In SMRL, data is represented by a keyword followed by an index number used to identify different data items. To keep SMRL simple, we refer to data by using functions (hereafter, *data functions*) with capitalized names (e.g., `Input(1)`). Table 2.1 presents the data functions in SMRL, grouped by category. Each data function returns a data class instance.

Fig. 2.3 presents the data model for Interaction and Output data. We focus on these two categories because they include the input and output types for the SUT. The other data categories concern data that is represented using Strings and used to assign values to Interaction data attributes. `Input` refers to a sequence of interactions between a user and the SUT; such interaction sequences are the only way to give inputs to the SUT. It

is consequently associated with `Action`, which represents an activity performed by a user. SMRL provides two types of actions: `WebAction` and `WaitAction`. `WebAction` captures a browser activity (e.g., sending an HTML form). It carries information about actions (e.g., the *URL* where the form is submitted and the *form inputs* provided to the URL with a POST request or the URL *parameters* for GET requests). `WaitAction` simulates time passing by changing the system time. `User` represents a system user.

In SMRL, source and follow-up inputs are always instances of the Input class, with the follow-up input derived through a Web-utility function (see Section 2.4.4). A follow-up input may differ from the source input because it includes a set of actions that differ from those in the source input or because it performs actions as a different user. In the rest of the chapter, to simplify the writing, we use the terms *follow-up action* and *follow-up user* to indicate an Action or a User derived by modifying an action or a user in the source input.

Instances of `OutputData` capture outputs generated by the system during a system-user interaction; each instance of `OutputData` is associated with an instance of `InteractionData`. Finally, both Actions and WebOutputs can be associated with a Session; indeed, MRs can modify the Session Cookies set before executing an Action and retrieve the Session Cookie returned by a Web page.

2.4.3 MR Operators

SMRL provides eight operators, i.e., `CREATE`, `IMPLIES`, `TRUE`, `FALSE`, `AND`, `OR`, `NOT`, and `EQUAL`. They facilitate the definition of *metamorphic expressions*, which are boolean expressions that must hold for an MR to be true. A *metamorphic expression* represents a specific type of `XBlockExpression` in XText. We use metamorphic expressions to decompose an MR into simple properties, and they are defined in a declarative manner, following standard practice in MT.

The MR in Fig. 2.2 contains a metamorphic expression using the operator `IMPLIES`. As the expression is within a loop body, the relation is valid only if the expression evaluates to true for all iterations over the input actions.

The semantics of operators `IMPLIES`, `AND`, `OR`, `TRUE`, `FALSE`, and `NOT` is straightforward. Operator `CREATE` defines a follow-up input by creating a copy of the source input passed as a second parameter. The follow-up input is identified by the keyword provided as the first parameter. In Fig. 2.2, operator `CREATE` defines the follow-up input `Input(2)` as a modified copy of `Input(1)` [47, 132]. Operator `CREATE` returns true if the follow-up input is successfully created. Depending on the context, operator `EQUAL` either evaluates the equality of two arguments or defines a follow-up input similarly to `CREATE`.

The construct `EQUAL(Input(2), Input(1))` enables writing an MR in a declarative manner without caring if `Input(2)` should be generated as a copy of `Input(1)` or if `Input(2)` already exists and it is equal to `Input(1)`. Operator `EQUAL` is evaluated to false when its first parameter refers to an input that has already been defined and used previously, in addition to not being equal to the second parameter. Operator `CREATE` in Fig. 2.2 can be

Table 2.2: Excerpt of the Web-specific functions in SMRL.

Function	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.
isResetPassword(Action action)	Returns true if the action is resetting the password.
EncodeUrl(String url)	Returns the `UTF_8` encoded version of the requested URL.
setChannel(String string)	Modifies the transfer protocol according to the string value (e.g. Http).
setSession(Session newSession)	Sets the session cookie value to match on the passed one.

replaced with `EQUAL` to obtain an equivalent MR (i.e., an MR that passes and fails with the same source inputs). The main difference between `CREATE` and `EQUAL` is that operator `CREATE` returns false if the identifier provided for the follow-up input already exists. Based on our experience, using operator CREATE simplifies the understanding of MRs for external readers.

2.4.4 Web-Utility Functions

MRs used in security testing often capture sophisticated Web systems properties that we cannot convey with arithmetic or simple boolean operators. Thus, SMRL is capable of providing some functions that capture standard Web system properties and modify Web data. Table 2.2 contains a portion of the 55 Web-specific functions in SMRL [55]. Each one is considered as a method of the SMRL API. Users can define additional functions as Java methods. The new functions can be incorporated into SMRL through the Xtext framework.

The MR in Fig. 2.2 uses the Web-specific functions `cannotReachThroughGUI`, `isError`, `isSupervisorOf`, and `changeCredentials`. The relation indicates that the same sequence of actions should provide different outputs when performed by two users under a condition: the two users cannot access one of the requested URLs by browsing the GUI of the system. In other words, if the system does not provide a URL to a user through its GUI, then she should not access the URL. Also, to avoid false alarms, the user who cannot access the URL from the GUI (indicated as `User(2)` in Fig. 2.2) should not be a supervisor with access to all the resources of the other user (`User(1)`). Finally, we avoid source inputs that return an error message to `User(1)` because it is impossible with these inputs to characterize the

CWE_266_267_268_269_285_522_529_862_863_OTG_AUTHZ_002.java

```
1  package smrl.mr.owasp;
2  import java.util.List;
6  @SuppressWarnings("all")
7  public class CWE_266_267_268_269_285_522_529_862_863_OTG_AUTHZ_002 extends MR{
8    public boolean mr() {
9      List<Action> _actions = Operations.Input(1).actions();
10     for (final Action action : _actions) {
11       {
12         ifThenBlock();
13         if (((!Operations.isSupervisorOf(Operations.User(), action.getUser())) &&
14           Operations.cannotReachThroughGUI(Operations.User(), action.getUrl())) &&
15           Operations.CREATE(Operations.Input(2), Operations.changeCredentials(Operations.Input(1), Operations.User()))) {
16           ifThenBlock();
17           boolean _OR = Operations.OR(
18             Operations.isError(Operations.Output(Operations.Input(1), action.getPosition())),
19             Operations.NOT(Operations.Output(Operations.Input(1), action.getPosition()).equals(
20                     Operations.Output(Operations.Input(2), action.getPosition()))));
21           if (_OR) {
22             expressionPass(); /* //PROPERTY HOLDS" */
23           } else {
24             return Boolean.valueOf(false);
25           }
26         } else {
27           expressionPass(); /* //PROPERTY HOLDS" */
28         }
29       }
30     }
31     return true;
32   }
33 }
34
```

Figure 2.4: Java code generated from the MR in Fig. 2.2 [47, 132].

output that should be observed for `User(2)`, which may be the same error, a different error, or an empty page [47, 132].

Function `cannotReachThroughGUI` in Fig. 2.2 checks if the URL of the current action cannot be reached from the GUI (Line 9). Function `isSupervisorOf` checks if `User(2)` is not a supervisor of `User(1)` (Line 10). Function `isError` returns true based on a configurable regular expression (Line 11) which checks if an output page contains an error message. Function `changeCredentials` creates a copy of a provided input sequence using different credentials. It is invoked to define the follow-up input (Line 12). 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. Given that this framework is developed in Java programming language, defining new functionalities is similar to Java. Thus, someone needs to write the function containing the required functionality and use its API in the MRs [47, 132].

2.5 Step 2: Transform MRs to Java

In Step 2, SMRL specifications are converted into Java code automatically. For this purpose, we extended the Xbase compiler (hereafter, SMRL compiler). Each MR is translated into a Java class with the relation name and package. The generated classes inherit from the `MR` class and implement its `mr` method.

The `mr` method executes the metamorphic expressions within the MR, returning `true` if the relation is satisfied and `false` otherwise. To acheive this, the SMRL compiler converts each boolean operator into a set of nested `IF` conditions. For instance, for `IMPLIES` operator, the generated code returns `false` if the first parameter is true and the second

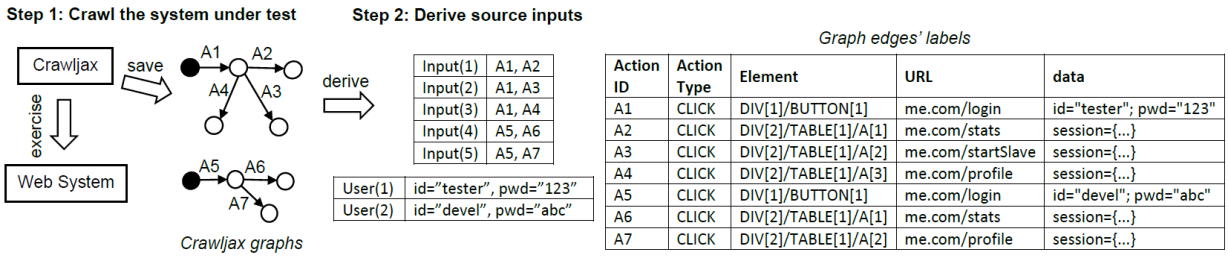

Figure 2.5: Data collection with a simplified example [47, 132].

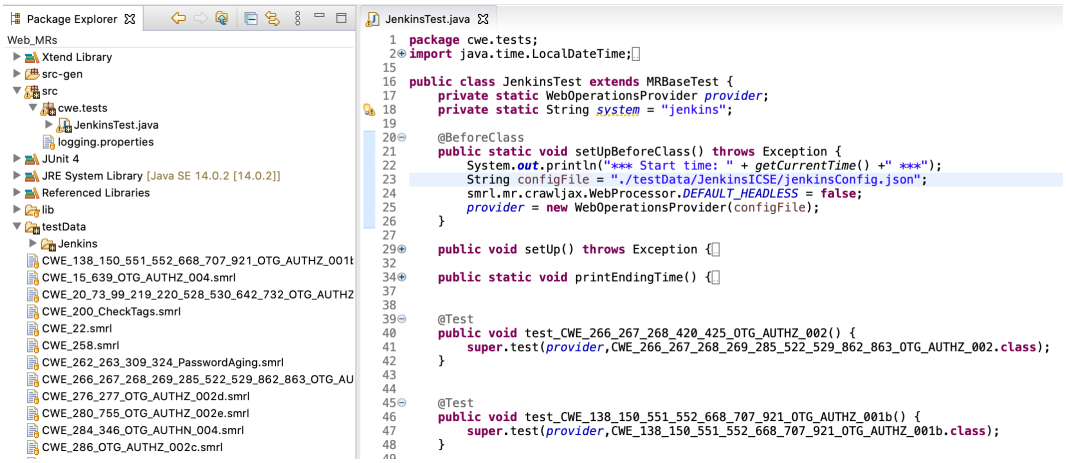

Figure 2.6: An example JUnit test case to select and execute MRs.

is false. When th MR is satisfied, the SMRL compiler generates a statement that returns `true` at the end of `mr` method.

Fig. 2.4 shows the Java code generated from the MR in Fig. 2.2. A loop control structure is derived from the loop instruction in the relation (Line 10). The loop body contains the Java code generated from the metamorphic expression using operator `IMPLIES` (Lines 13-28). The first `if` condition checks whether the first parameter of operator `IMPLIES` holds (Lines 13-15). The nested `IF` block examines whether the second parameter of `IMPLIES` holds (Line 21). If the expression does not hold, `mr` returns `false` (Line 24). 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` erase the generated follow-up inputs at each iteration and keep track of the last output observed. Function `ifThenBlock` determines if we are within the first `ifThenBlock` of the MR (it happens when `ifThenBlock` is invoked before the first metamorphic expression of the MR) and, in the affirmative case, erases the follow-up inputs generated so far. Indeed, the first `ifThenBlock` function is executed at the beginning of each iteration. Function `expressionPass` empties the list of inputs processed by the last metamorphic expression. Function `Output` tracks these inputs to provide engineers with contextual information in the case of a failure [47, 132].

2.6 Step 3: Data Collection

In Step 3, we rely on an extended version of the Crawljax Web crawler to automatically derive source inputs [143, 145]. 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 whose edges capture the action performed to reach a given state (e.g., clicking on a button). Crawljax detects the system states based on the content of the displayed page. Our extension relies on the edit distance to distinguish the system states [121]. We keep a cache of the HTML page associated with 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 with 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 when a timeout is reached [47, 132].

Our Crawljax extensions allow replicating and altering portions of a crawling session, which is necessary to simulate attacks (e.g., by changing the URL accessed within an input sequence). Thus, in addition to (i) the XPath of the elements targeted by the actions (e.g., a button clicked on) and (ii) the Crawljax actions, our extension records (iii) the requested URLs by the actions, (iv) the data in the HTML forms, and (v) the background URL requests. This additional information enables replicating modified sections of crawling sessions, even for request URLs not present on the last Web page returned by the system. To initiate crawling, only the SUT's URL and a list of credentials are required.

Fig. 2.5 illustrates the steps involved in the data collection. First, Crawljax generates the graphs of the system under test. Second, our toolset automatically extracts source inputs from the graphs. A source input is a path from the root to a leaf of a Crawljax graph using a depth-first traversal. Third, the SMRL functions query the source inputs (see Section 2.7). For instance, `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, *MST-wi* also processes manually implemented test scripts to generate additional source inputs. More precisely, it processes scripts based on the Selenium framework [214] and derives a source input from each script. We utilize test scripts to exercise complex interaction sequences not triggered by Crawljax (see Section 2.10). Crawljax conducts an exhaustive exploration of the Web interface, which is often not achieved by test scripts. Users can define new scripts or reuse the ones developed for functional testing.

2.7 Step 4: Execute the MT Framework

We conduct automated testing based on the executable MRs in Java and the data collected by the data collection framework (Step 4 in Fig. 2.1). Our testing framework is built on the JUnit framework [4], enabling integration of MT into traditional testing environments.

```
input  MR, the bytecode of the metamorphic relation to be executed
input  dataProvider, an object that exposes the data collected by the crawlers
output Failures, a list of failing executions with contextual information
 1: srcTypes ← extractSourceInputTypes(MR)
 2: iterateOverInputTypes(MR, dataProvider, 0, dataTypes)
 3: return Failures

 4: function ITERATEOVERINPUTTYPES(MR, dataProvider, i, dataTypes)
 5:     while dataProvider.hasMoreViews(dataTypes[i]) do
 6:         dataProvider.nextView(dataTypes[i])
 7:         if (i < dataTypes.length) then   //need to iterate over other types
 8:             iterateOverInputTypes(MR,dataProvider, i+1,srcTypes)
 9:         else  //we have set a view for every input type in the relation
10:             result = MR.run() //execute the metamorphic relation
11:             if ( result == false) //the MR does not hold
12:                 addFailure(Failures,dataProvider) //trace the failure
```

Figure 2.7: Metamorphic testing algorithm.

To choose MRs to be executed, the engineer or user can write a JUnit test case. Fig. 2.6 illustrates a JUnit test case selecting multiple MRs (i.e., CWE_266..._OTG_AUTHZ_002, and CWE_138..._OTG_AUTHZ_001b) through function `test`. MRs must be copied into the workspace (see the files with extension `.smrl` in the project explorer window in Fig. 2.6) and referred to in the JUnit test case (see JenkinsTest.java in Fig. 2.6). *MST-wi* provides a Java class, `MRBaseTest` (Line 16 in Fig. 2.6), which extends the JUnit framework with utility functions to simplify MRs selection. The `setUpBeforeClass` method is used to define the configuration for `WebOperationsProvider` (Lines 21-26), i.e., the component responsible for loading source inputs. The MRs are executed as standard JUnit test classes, either through the console wrapper or the Eclipse user interface. Each MR is handled as a distinct JUnit test case (Lines 40-48 in Fig. 2.6). For each MR, `MRBaseTest` class automatically invokes our MT algorithm that executes the MR.

Fig. 2.7 presents our MT algorithm. The algorithm takes as input an 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*). To do so, `extractSourceInputTypes` (Line 1) identifies the calls to the *data representation functions* using the ASM static analysis framework [12]. Function `iterateOverInputTypes` (Line 2) ensures that each source input is used in at least one execution and that all possible source input combinations are stressed during the execution of the relation (e.g., all available URLs with all configured users). It iterates on all available items for a given input type (e.g., all available users). It is invoked recursively for each input type in the MR.

The function `iterateOverInputTypes` is driven by the methods provided by the data provider (Lines 5 and 6). The data provider functions works as a circular array, delivering a different view of the collected data in each iteration of `iterateOverInputTypes`. For N input items of a given type (e.g., User), function `nextView` (Line 6) generates N different views with items shifted by one position. The MR is executed (Line 10) after obtaining the views. The algorithm generates follow-up inputs from the source inputs at each invocation of operator `EQUAL`. For example, operator `EQUAL` in Fig. 2.2 makes `Input(2)` refer to a copy of the input sequence returned by function `changeCredentials`. Function `addFailure` stores the failure context information (i.e., source inputs, follow-up inputs, and system

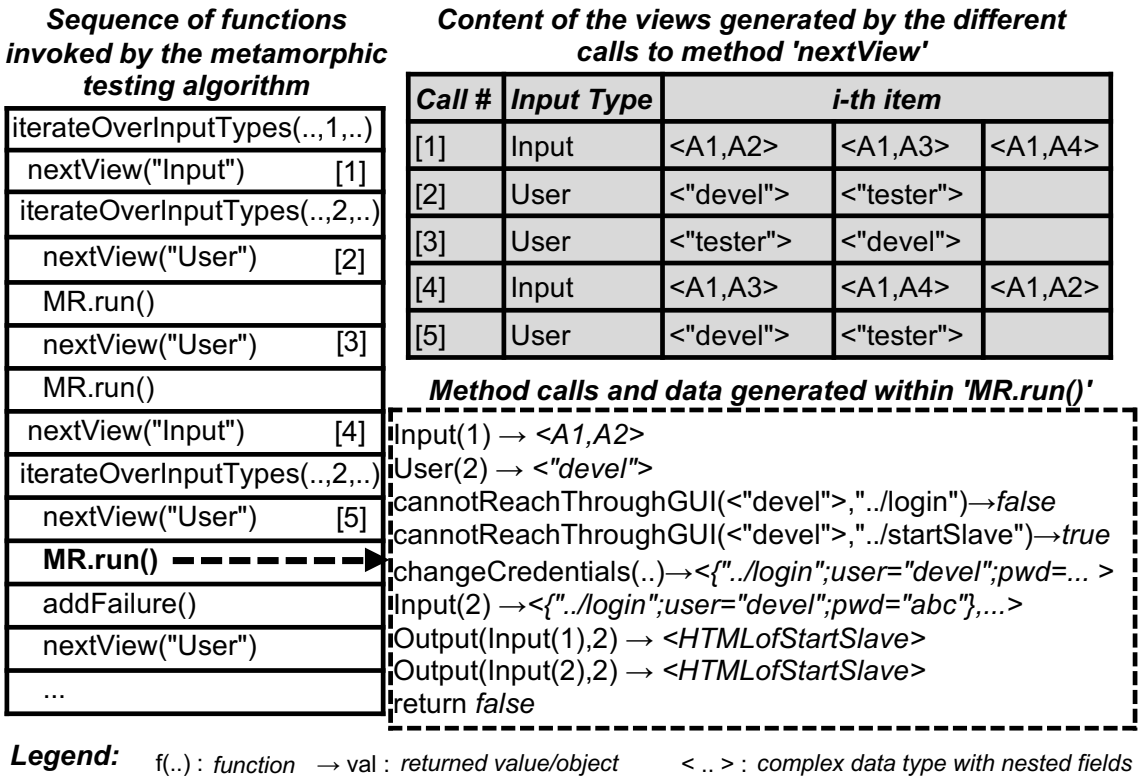

Figure 2.8: Example data processing for the relation in Fig. 2.2 [47, 132].

outputs) when the MR does not hold (Lines 11 and 12). *MST-wi* reports only failures that perform HTTP requests (e.g., accessing a URL) not generated by input sequences that led to previously reported failures. Therefore, it reduces the time spent to manually analyze failures triggered by distinct follow-up inputs exercising the same vulnerability. To guarantee that all input item combinations are used, function `nextView` is iteratively invoked until all the items of a given input type are processed (Line 5). The *MST-wi* data representation function *RandomValue* (see Table 2.1) returns a random data value. For scalability reasons, it is configured to generate up to 100 different values, thus leading to 100 views.

Fig. 2.8 presents the execution of the relation in Fig. 2.2. The table on the left illustrates the sequence of functions invoked by our algorithm. In this example, two views for `User` are examined for each view of `Input`. The first two invocations of `MR.run` return true (not shown in Fig. 2.8) because the *login* and *stats* pages have been accessed by both users *devel* and *tester*, thereby satisfying the implication. The third invocation of `MR.run` returns false, as the output page for the *startSlave* URL is identical for both input sequences and, therefore, the relation does not hold. We rely on edit distance to assess if Web pages are identical.

2.8 Catalog of Metamorphic Relations

We derived a catalog of MRs from the activities described in the OWASP book on security testing [146] and from a set of security vulnerability types related to violations of security design principles and reported in the Common Weakness Enumeration (CWE) database [14].

The OWASP book on security testing presents detailed descriptions of 90 testing activities (hereafter *OWASP security testing activities*) for Web systems; each OWASP security testing activity targets a specific vulnerability type. The activities provide information for implementing metamorphic relations. For example, for the bypass authorization schema vulnerability, OWASP suggests collecting links in administrative interfaces and directly accessing the corresponding URLs by using the credentials of other users [47, 132]. Using this suggestion, we defined the MR in Fig. 2.2.

The CWE vulnerability types considered for our catalog concern violations of security design principles. As further discussed in Section 2.9, they are the ones we selected to address our research questions about the applicability of *MST-wi* as they correspond to system-level violations resulting from design mistakes.

We can perform security testing activities (e.g., simulate attacks) in multiple ways and may have more than one relation for each OWASP activity or CWE vulnerability type. Also, not all testing activities benefit from MT. We discuss the capabilities of MT in Section 2.9.

Our catalog includes 76 MRs in total. 22 MRs automate 16 OWASP activities; 54 MRs detect 101 CWE vulnerability types. 15 MRs both automate OWASP activities and detect CWE vulnerability types, which is unsurprising since OWASP activities aim to verify that secure design principles (e.g., correctly implement and configure authorization mechanisms) are in place. Overall, our MR catalog discovers 102 security vulnerability types, including 101 belonging to the CWE catalog (some of which are discovered by OWASP testing activities) and one vulnerability type being targeted by OWASP testing activities only.

We perform security testing 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 [250]. However, we do not rely only on random values to obtain valid inputs that match the system configuration. Instead, we modify source inputs using the data provided by the SMRL Web-utility functions, which return domain-specific information (e.g., protocol names), random values, and crawled data. Finally, by capturing properties of the output generated from the source and follow-up inputs, we identify vulnerabilities that cannot be detected with implicit test oracles [43] (e.g., crashes).

We present some of the MRs in our catalog in Figs. 2.10 to 2.19. The entire catalog is available for download [55]. All the MRs in our catalog follow the template in Fig. 2.9

```
1:  Iterate over all the Actions of Input
2:    (optional) Iterate over all the Actions of another Input
3:    (optional) Iterates over all the parameters, form inputs, or session cookies of the selected Action
4:    IMPLIES (
5:      [a precondition holds ]repeatable multiple times
6:      [ and a follow-up input is successfully created]repeatable multiple times
7:      [ and the follow-up input is successfully modified ]repeatable multiple times
8:    ,
9:      condition on the outputs generated for the source and follow-up inputs
10:   )
```

Figure 2.9: Template common to all the MRs in our catalog.

in Section 2.8.1. Also, the sequences of invocations of Web-utility functions in our MRs can be grouped into a set of patterns given in Section 2.8.2. The identification of patterns facilitates the definition of solutions to a problem [30]; in our case, the MR patterns facilitate the identification of MRs to address the problem of automatically performing security testing based on the description of a vulnerability type and corresponding attacks. In other words, MR patterns help understand how to derive MRs that simulate certain attacks; in turn, they enable engineers to define new MRs for attacks not considered yet (e.g., by looking at similarities with attacks already considered).

2.8.1 Metamorphic Relations Template

This section presents the template followed by all the MRs in our catalog. This template was not defined up-front but is the result of deriving MRs for more than 100 specifications (16 OWASP activities and 101 CWE vulnerabilities) and analyzing their commonalities.

All the MRs include a loop (Line 1 in Fig. 2.9) to define multiple follow-up inputs by iteratively modifying the actions of the source input. MRs may have additional nested loops used either to combine multiple source inputs (Line 2) or to iterate over all the parameters, form entries, or session cookies belonging to the Web page on which the action is performed (Line 3). For example, `CWE_287a_425_OTG_AUTHN_001` (Fig. 2.10) works with all the login actions observed in the source input. Function `isLogin`() returns true only if the current action performs a login; otherwise, the implication trivially holds, and no follow-up input is generated.

We express all our MRs using an implication (operator `IMPLIES` in Line 4 of Fig. 2.9). The left-hand side of the implication often starts with verifying if a precondition holds (Line 5). Then we check if the follow-up input is successfully defined (Line 6). Finally, we ensure that the follow-up input can be further modified through multiple function calls appearing on the left-hand side of the implication (Line 7). For instance, the first and second follow-up inputs in MR `CWE_302_471_472_784_807` (Fig. 2.11) are for performing the actions of the source inputs with another user (to ensure the follow-up user cannot perform them) and the attack, respectively.

The right-hand side of the implication usually captures the relation between the outputs of the source and follow-up inputs. For instance, according to `CWE_287a_425_OTG_AUTHN_001` in Fig. 2.10, the output for the follow-up input (which performs a login on the unencrypted

HTTP channel) should be different from the output for the source input because it should not be possible to login using the HTTP channel.

In the following, we describe the elements of our template: preconditions, generation of follow-up inputs, and output conditions (postconditions). Table 2.3 provides an overview of possible template element instances.

Table 2.3: MRs template element instances

Template element	**Element instance**
Preconditions	User precondition
	Action precondition
Generation of follow-up inputs	Same user
	Different user
	Same actions
	Actions subset
	Added action(s)
	Modified Action(s)
Output conditions (postconditions)	Verify equality
	Verify difference
	Verify other predicate

Preconditions

Preconditions generally concern the actions in the source input or the user performing these actions.

User preconditions are applied to identify the users performing the follow-up inputs based on access level. For instance, we use the function *isSupervisorOf()* to ensure that the follow-up user doesn't have access to a resource from another user. Also, we rely on the function *notTried(User, actionURL)* to avoid testing the same URL multiple times with the same user; indeed, this function indicates if we have already exercised a URL with a follow-up input for the specified user.

Action preconditions avoid redundant follow-up inputs and false positives. For example, the function *notTried(actionURL)* is used to not test the same URL twice (regardless of the user performing the action) in MRs that do not target authorization and authentication vulnerabilities. To reduce false positives, we may avoid actions whose output leads to error or alert messages. For instance, `!Output(Input(1),pos).hasAlert` for `CWE_79_a_XSSreflected` (Fig. 2.12) checks if the source input action does not lead to a popup message, i.e., the condition used to determine if the XSS injection attack is successful.

72 MRs (94%) and 33 MRs (43%) in our catalog include an action precondition and a user precondition, respectively.

```
MR CWE_287a_425_OTG_AUTHN_001 {
{
   for ( Action action : Input(1).actions() ) {                        //(1)
      var pos = action.getPosition();                                  //(2)
      IMPLIES(
          isLogin(action) &&                                           //(3)
          notTried(action.url) &&                                      //(4)
          CREATE ( Input(2), Input(1) ) &&                             //(5)
          Input(2).actions.get(action.position).setChannel("http")     //(6)
          ,
          different ( Output(Input(1),pos),  Output(Input(2),pos) )    //(7)
    );//end-IMPLIES
   }//end-for
  }
 }//end-MR
}//end-package
```

(1) For loop iterates over all actions of Input(1). (2) Stores the parameters of the current action in a variable. (3) It checks that the action is a login. (4) It verifies that this login URL was not already tested. (5) Creates the follow-up input by copying the input(1). (6) Sets the HTTP channel for the follow-up input. (7) A login operation should not succeed if performed on the HTTP channel. checks if the output generated by the login operation is different in the two cases.

Figure 2.10: CWE_287a_425_OTG_AUTHN_001: Testing for credentials transported over an encrypted channel

Generation of follow-up inputs

Follow-up inputs can be generated via input generation strategies focusing on the sequence of actions in the source input and the user performing actions.

We have two strategies for the *user*: (i) keeping the same user of the original source input sequence and (ii) selecting another user. The second one is adopted for MRs looking for vulnerabilities related to confidentiality, authorization, or authentication (e.g., accessing a resource with different users). The first one is used in all the other MRs. In total, eight MRs in our catalog create follow-up inputs whose actions are performed with a user different than the one in the source input.

Concerning actions, the strategies employed to derive follow-up inputs include (i) relying on the same sequence of actions of the source input sequence, (ii) retaining only a subset of the actions, (iii) adding actions to the source input, and (iv) modifying the actions in the source input.

We *keep the same sequence of actions* of the source input for MRs that focus on authorization and authentication vulnerabilities; indeed, to test the authorization mechanisms of a system, we can perform the same actions of the source input but with a different access level (e.g., `CWE_266_..._OTG_AUTHZ_002` in Fig. 2.2). Eight MRs in our catalog (10%) perform the same actions of the source input either with a different user (e.g., `CWE_266_..._OTG_AUTHZ_002`) or in an alternative channel (e.g., `CWE_287a_425_OTG_AUTHN_001` in Fig. 2.10).

We take a *subset* of the actions in the source input to speed testing up or ensure that the system enforces the precedence relation between the actions (if there are any). An

example MR of the first case is CWE_286_OTG_AUTHZ_002c (Fig. 2.14). This MR tests the condition that if a user navigating the GUI cannot access a URL, the same URL should not be available to the same user when she directly requests it from the server. It relies on a subset of the source input actions since, to test the condition above, it is sufficient to create a follow-up input including only the action accessing the reserved URL rather than performing all the actions. The system is vulnerable if the follow-up input leads to a successful retrieval of the resource pointed to by the URL (i.e., the same output as the source input action). An example of the second case (i.e., ensuring that precedence relations are enforced) is that of a system that should validate a user session to ensure that she has confirmed her email address after signing up. The system is vulnerable if the user can log in without confirming her email address. The source input sequence includes the registration to the service, the email confirmation, and a successful log-in. The follow-up input should cover only the registration action and the log-in without confirming the email address. If the log-in is successful, the system is vulnerable as it should ensure that the user is registered with a valid email address. Note that *MST-wi* automatically generates such follow-up inputs by iteratively generating distinct subsets of actions from the source input sequence (see MR CWE_841 in Fig. 2.16). Five MRs (7%) in our catalog are obtained by selecting a subset of actions from follow-up inputs.

We *add* one or more actions to a source input to test scenarios when we expect such a change in action sequence to lead to different results. Usually, the actions added are user actions belonging to a source input (not necessarily the one used to define the follow-up input) or actions capturing environmental factors (e.g., the passing of time). An example user action copied into the follow-up input is given in MR OTG_SESS_003 (Fig. 2.17); OTG_SESS_003 scans the source inputs to identify a sign-up action to be copied into the follow-up input, after the login. In OTG_SESS_003, the presence of a sign-up action after the login enables us to verify that the session ID is updated after every sign-up. An example environment action added to a source input sequence is given in CWE_262_263_309_324 (Fig. 2.13), which tests the system's password aging mechanism through a *DelayAction*. We consider a source input including login and add a *DelayAction* that changes the system's date to next year, thus simulating the passing of time. When the user performs the next action in the sequence, the system should ask the user to change his credentials instead of completing the requested action and returning the same results as the source input. We obtain the follow-up inputs in nine MRs by adding actions to the source input.

We *modify* actions by changing the assignments to parameters that are inputs of the SUT (i.e., URL parameters, form entries, session cookies, certificates) or that control the communication channel (i.e., encryption algorithms, communication protocol, HTTP method). These changes may be applied to follow-up inputs that match the source inputs or include other differences (e.g., subsets). For example, to test a system for code injection vulnerabilities, we execute the same sequence of actions as in the source inputs and modify form entries by relying on known attack vectors (e.g., concatenating SQL injection commands to the original input values). An example MR that involves the modification of a communication protocol is CWE_287a_425_OTG_AUTHN_001 in Fig. 2.10. In 57 MRs of our catalog, the follow-up inputs perform the same sequence of actions as the source input but with at least one parameter modification.

Output conditions

Output conditions (postconditions) in our MRs check if the outputs of the source and follow-up inputs (or a subset of their actions) are equal, different, or satisfy a predicate (e.g., the output is erroneous or includes information that has already been observed by the user) implemented by an SMRL function.

SMRL MRs capture properties that should hold if the system is not vulnerable. Therefore, we expect the *same output* for the source and follow-up inputs when the attack captured by the follow-up input should not alter the system behavior, that is, when the system is supposed to detect an attack vector and ignore its effects. For example, the SUT should sanitize the received inputs by removing the code injection attack vector added to the original input and return the same output as the source input. In 20 of our 76 MRs (26%), the follow-up input is supposed to lead to the same output as the source input.

In contrast, for 30 MRs in our catalog (39%), the outputs generated by the follow-up inputs are expected to be *different* from those of the source inputs. For instance, in MR `CWE_266_..._OTG_AUTHZ_002` (Fig. 2.2), accessing a resource dedicated to the source input's user should lead to a different output (e.g., the home page or an error page).

Last, we can also verify *predicates* on the generated outputs. Predicates are used, for example, to verify that the returned output should be accessible by the user in the MRs `CWE_20_..OTG_AUTHZ_001a` and `CWE_15_639_OTG_AUTHZ_004`. MR `CWE_20_..OTG_AUTHZ_001a` (Fig. 2.20) replaces URL parameters (e.g., an ID) with paths of files on the SUT. Similarly, `CWE_15_639_OTG_AUTHZ_004` (Fig. 2.15) replaces URL parameter values with values observed only with other users. Though we cannot predict the effect of altering URL parameters — we cannot know in advance if the value is legal for the user performing the action — the user should be able to access the output when browsing the GUI. Since the crawler browses the GUI with the different users, function `userCanRetrieveContent` checks if the output has already been observed with any source input collected by the crawler. In our catalog, 61 MRs include at least one predicate on the generated outputs.

2.8.2 Metamorphic Relation Patterns

This section presents how the instances of the template elements are combined to form the MR patterns. A pattern is captured by a set of element instances in one or more MRs. Our focus on the common set of element instances across MRs aims to enable the systematic identification of patterns while organizing them in a taxonomy. Further, with patterns, the reader can more easily compare MR characteristics. For example, at the end of this section, we discuss why some patterns are less frequent than others. Covered element instances provide us with a systematic approach to identify patterns. The following conditions determine our patterns: (1) the user in the follow-up inputs should either match or differ from the one in the source input, (2) in the follow-up inputs, we should observe a sequence of action that contain either the same sequence of actions as the source inputs, or additional actions, or a subset of source inputs' actions, or modified actions.

To identify MR patterns, we first filled in a mapping table tracing MRs to the template elements in the previous section. Then, we grouped the MRs covering the same combination of elements. By definition, an MR can implement only one pattern. The resulting MR patterns are reported in Table 2.4.

In total, we identified 23 patterns. Six patterns are implemented by at least five MRs, and twelve patterns are implemented by only one MR. In the following, we discuss the three most frequent patterns.

Table 2.4: MR Patterns

Pattern ID	Preconditions		Generation of follow-up inputs						Output condition			Number of MRs
	User precondition	Action precondition	Same user	Different user	Same actions	Actions subset	Added action(s)	Modified action(s)	Verify equality	Verify difference	Verify other Predicates	
P1	1	1	1	0	0	0	0	1	0	0	1	13
P2	0	1	1	0	0	0	0	1	1	0	1	13
P3	0	1	1	0	0	0	1	0	0	1	0	7
P4	0	1	1	0	0	0	0	1	0	0	1	6
P5	0	1	1	0	0	0	0	1	0	1	1	5
P6	1	1	1	0	0	0	0	1	0	1	1	5
P7	1	1	1	0	0	0	0	1	1	0	1	4
P8	0	1	1	0	0	0	0	1	0	1	0	4
P9	1	0	1	0	0	0	0	1	0	0	1	3
P10	1	1	0	1	1	0	0	0	0	1	1	2
P11	0	1	1	0	1	0	0	0	0	0	1	2
P12	0	0	1	0	0	0	1	0	1	0	0	1
P13	1	1	0	1	0	1	0	0	0	1	1	1
P14	0	1	0	1	1	0	0	1	0	1	0	1
P15	0	1	1	0	0	1	0	0	0	1	1	1
P16	0	1	1	0	0	1	0	1	0	1	1	1
P17	0	1	1	0	1	0	0	0	0	1	0	1
P18	0	1	1	0	1	0	0	1	0	0	1	1
P19	1	1	0	1	0	0	0	1	0	1	1	1
P20	1	1	0	1	0	0	1	0	0	0	1	1
P21	1	1	0	1	0	1	0	0	1	0	0	1
P22	1	1	1	0	0	1	0	0	0	1	1	1
P23	1	1	0	1	1	0	0	0	1	0	1	1

Two patterns occur with the same frequency (i.e., *P1* and *P2* in Table 2.4); we present them in the order they appear in Table 2.4. The first pattern (i.e., *P1*) includes the following template elements: *user precondition(s), action precondition(s), same user, modified action(s), and verify other predicates*. It is implemented by 13 MRs and used to ensure that a user cannot retrieve protected resources by providing crafted data as input.

Indeed, the same user performs the same actions as in the source input sequence after modifying some of them. User and action preconditions enable the selection of cases where the MR should hold. All these 13 MRs include an action precondition verifying that the URL is tested only once (to speed testing up). Further, 12 of these 13 MRs include a user precondition checking if the user has already retrieved the content of the source input (to avoid testing with non-deterministic sequences); one MR verifies that the user performing the follow-up actions is not an administrator since she might have the permission to perform these actions.

The predicates ensure that the generated output is either an error page or content that the user is supposed to be able to access (i.e., it has been accessed during crawling or functional testing as reported by functions *userCanAccess* or *userCanRetrieveContent*). After providing a code injection string, `CWE_94_96_B` (Fig. 2.18), for instance, verifies that a user can retrieve only an error page or a resource that she can access.

The other most frequent pattern (i.e., *P2*) is similar to the first one, except it does not verify user preconditions and verifies output equality. It includes the following template elements: *action precondition, same user, modified action(s), verify equality, and verify other predicates*. It is used to simulate attacks where the user provides a crafted input (e.g., invalid character) that should be either sanitized (the same output is returned) or lead to an error page (this last condition is verified with a dedicated predicate on the output). An example MR of this pattern is MR CWE_792_793_794_795_796_797_A (Fig. 2.19).

The third pattern (i.e., *P3*) includes the following template elements: *action precondition(s), same user, added action(s), and verify the difference*. It is implemented by seven MRs where a follow-up input with a different number of steps should lead to different outputs (e.g., *OTG_SESS_003* in Fig. 2.17, which duplicates a signup action and verifies that the action leads to a session cookie different from the cookie of the previous signup action).

Twelve patterns are implemented by only one MR. Six of these patterns result from MRs creating multiple follow-up inputs, which leads to combinations of template elements normally not appearing together (e.g., equal and subset). Another pattern (i.e., *P12*) concerns an MR that does not verify any precondition (it generates follow-up inputs from all the available source inputs) because discovering the vulnerability likely depends on the system state, the sequence of actions previously executed, or the user performing the action; it differs from all the other patterns because they all include at least one precondition. The presence of state-dependent vulnerabilities is however rare (e.g., the successful exploitation of a code injection vulnerability is unlikely to depend on the actions performed before), which is why all our patterns, except one, have at least one precondition.

```
MR CWE_302_471_472_784_807{
{
   for (Action action: Input(1).actions()){                          //(1)
      var pos = action.getPosition();                                //(2)
      var session = Output( Input(1), pos).session as CookieSession;//(3)
      CREATE( Input(2), changeCredentials(Input(1), User()) )        //(4)
      var session2 = Output(Input(2), pos).session as CookieSession;//(5)
      var notTried = notTried(action.url,Input(1).actions().get(pos).
      getElementURL())                                               //(6)
      var mappings = session.keyValueMappings.entrySet;              //(7)

      for ( Entry<String,String> cookie : mappings){                 //(8)
         var type = typeOf(cookie.value);                            //(9)
         IMPLIES(
             notTried &&
             type == Boolean &&                                      //(10)
             cannotReachThroughGUI( User(), action.url ) &&          //(11)
             different(Output(Input(1), pos) ,Output(Input(2), pos))
             &&                                                      //(12)
             CREATE ( Input(3), Input(2) )  &&                       //(13)
             Input(3).actions.get(pos).setSession( session2 ) &&     //(14)
             session.setCookie(
                 new Cookie(cookie.key, String.valueOf(!Boolean.valueOf(
                 cookie.value))))                                   //(15)
           ,
         OR(                                                       //(16)
             different(Output( Input(1), pos) , Output(Input(3), pos)),
             isError(Output(Input(3) , pos)))
     );//end-IMPLIES
    }//end-for
   }//end-for
  }
 }//end-MR
}//end-package
```

(1) For loop iterates over all actions of the Input(1). (2) Stores the parameters of the current action in a variable. (3) Saves the cookie value of the Input(1) in a variable. (4) Creates a follow-up input with different credentials. (5) Saves the cookie value of the follow-up user in a variable. (6) To speed up the process, verifies that the element URL has not been tried before. (7) Stores the session key values in a variable. (8) For loop iterates over the session values. (9) Assigns the type of the cookie to a variable. (10) Filters all cookie types other than Boolean (11) Makes sure the URL is not accessible without login. (12) To avoid False positives, filters the actions with the same output for Input(1) and Input(2). (13) Creates the follow-up input, named Input(3). (14) Sets the session2 as the session value of the Input(3). (15) Flip the value of the Boolean cookie. (16) The system should give a different output or should show an error.

Figure 2.11: CWE_302_471_472_784_807: Testing for assumed-immutable elements

```
 MR CWE_79_a_XSSreflected {
{
  keepDialogsOpen = true;                                          //(1)
  for ( Action action : Input(1).actions() ) {                     //(2)
     for ( var x = 0; x < action.parameters.size; x++){            //(3)
        var pos = action.getPosition();                            //(4)
        IMPLIES(
              notTried(x+action.url, Input(1).actions().get(pos).
              getElementURL()) &&                                  //(5)
              !Output(Input(1), pos).hasAlert &&                   //(6)
              CREATE( Input(2), Input(1) ) &&                      //(7)
              Input(2).actions().get(pos).setParameterValue(x,
              XSSInjectionString())                                //(8)
              ,
           OR(                                                     //(9)
               Output(Input(2), pos).emptyFile,
              !Output(Input(2), pos).hasAlert)
     );//end-IMPLIES
    }//end-for
   }//end-for
  }
 }//end-MR
}//end-package
```

(1) It avoids clicking on OK; because dialogs are normally ignored by our framework by clicking on OK. (2) For loop iterates over all actions of the Input(1). (3) The second loop iterates over all parameters of the action. (4) Defines a variable to store the position of the Input(1)'s action. (5) Checks if the parameter of the action URL has not seen before, to speed up the process. (6) Verifies that the action does not originally contain any alert. (7) Creates the follow-up input by copying the Input(1). (8) Sets the injected XSS string as the parameter value of the follow-up input. (9) Checks either the attack was not performed or no effect shall be observed (the effect of the XSS is usually visualized when reaching the page where we inject the XSS).

Figure 2.12: CWE_79_a_XSSreflected: Testing for reflected cross-site scripting

```
MR CWE_262_263_309_324 {
{
   for ( var x=0; x < Input(1).actions().size() ; x++ ){           //(1)
       IMPLIES (
           isLogin( Input(1).actions().get(x) ) &&                 //(2)
           !isError( Output(Input(1),x+1) )&&                      //(3)
           CREATE( Input(2) , Input(1) ) &&                        //(4)
           Input(2).addAction( x, Wait( 60*60*24*30*12*1000) )     //(5)
           ,
           different( Output(Input(2),x+2) ,  Output(Input(1),x+1)) //(6)

    ); //end-IMPLIES
   } //end-for
  }
 } //end-MR
} //end-package
```

(1) For loop iterates over the actions of Input(1). (2) Checks if the current action x is doing "log in". (3) Checks that the action-x of the Input(1) does not include any errors. (4) Creates the follow-up Input by copying the Input(1), named Input(2). (5) Add a wait action to Input(2) that moves time forward for a year i.e. the expected time for resetting the password. (6) Based on the above description the results should be different; it shall prompt a password update request.

Figure 2.13: CWE_262_263_309_324: Testing for password aging

```
MR CWE_286_OTG_AUTHZ_002c {
{
  for(var y = Input(1).actions().size()-1; ( y > 0 ); y--){        //(1)
      IMPLIES(
          (!isSupervisorOf(User(), Input(1).actions().get(y).user)
          ) &&                                                     //(2)
          afterLogin(Input(1).actions().get(y)) &&                 //(3)
          cannotReachThroughGUI(User(), Input(1).actions().get(y).
          getUrl()) &&                                             //(4)
          CREATE(Input(2), Input(LoginAction(User()), Input(1).
          actions().get(y)))                                       //(5)
          ,
       OR(                                                         //(6)
          isError(Output(Input(1), y)),
          different(Output(Input(1), y),Output(Input(2), 1)))
  ); //end-IMPLIES
  } //end-for
 }
} //end-MR
} //end-package
```

(1) The for loop iterates over all the actions of an input sequence. (2) Checks whether the user in User() is not a supervisor of the user performing the y-th action. (3) Verifies that the y-th action is performed after a login. (4) Verifies that the follow-up user cannot retrieve the URL of the action through the GUI (based on the data collected by the crawler). (5) Defines a follow-up input that performs the login as the follow-up user (6) The system checks if the y-th action from the source input leads to an error page Or the output generated by the action containing the URL indicated above, lead to two different outputs in the two cases.

Figure 2.14: CWE_286_OTG_AUTHZ_002c: Testing for incorrect user management

```
MR CWE_15_639_OTG_AUTHZ_004 {
{
   for ( Action action : Input(1).actions() ){                     //(1)
      for ( var par=0; par < action.getParameters().size()
            && notTried( action.getUser(), action.url); par++ ){   //(2)
        for( usedValue : parameterValuesUsedByOtherUsers(action, par
        )) {                                                       //(3)
          var pos = action.getPosition();                          //(4)
          IMPLIES(
             CREATE ( Input(2), Input(1) ) &&                      //(5)
             Input(2).actions().get(pos).setParameterValue(par,
             usedValue)                                            //(6)
             ,
          OR(                                                      //(7)
             Output(Input(2),pos).isError()
             ,
             userCanRetrieveContent( action.user, Output(Input(2),pos) ))
      );//end-IMPLIES
     }//end-for
    }//end-for
   }//end-for
  }
 }//end-MR
}//end-package
```

(1) The first loop iterates over all the actions of the input sequence. (2) The second iterates over all the parameters of the action. (3) The third loop iterates over all used values by other users. (4) Stores the parameters of the current action in a variable. (5) Defines the follow-up input. (6) Sets a parameter value to a value that is used by other users. (7) Checks if the content of the output is either an error message Or some content that can be retrieved from the GUI.

Figure 2.15: CWE_15_639_OTG_AUTHZ_004: Testing for externally controlled elements

```
MR CWE_841 {
{
   for( var x = 0; x < Input(1).actions().size(); x++ ){              //(1)
      for( var y = x+2; isSignup(Input(1).actions().get(x)) &&
           (y < Input(1).actions().size()); y++){                    //(2)
         IMPLIES (
             isLogin(Input(1).actions().get(y))&&                    //(3)
             CREATE ( Input(2) , Input ( Input(1).actions().subList(y,
                     Input(1).actions().size()))  &&                 //(4)
             Input(2).addAction(0,Input(1).actions().get(x))   &&    //(5)
             Input(2).addAction(0, new ResetSUTAction())             //(6)
             ,
             AND(different(Output(Input(2),1),  Output(Input(1),y)) //(7)
                ,
                  isError(Output(Input(2),1)))
     ); //end-IMPLIES
    } //end-for
   } //end-for
  }
 } //end-MR
} //end-package
```

(1) The first loop iterates over all the actions to find a signup action. (2) The second loop iterates over all the actions to find a login action performed after the signup. (3) Verifies that the action y is a login. (4) Creates the follow-up input with the sequence of actions after the login (action y). (5) Makes sure that we start with a signup action (6) Reset the actions for next iteration (7) Verifies that the system gives two different outputs or will give an error by executing the follow-up input.

Figure 2.16: CWE_841: Testing for improper enforcement of behavioral workflow

```
MR OTG_SESS_003 {
{
   for( Action signup : Input(1).actions() ){                         //(1)
     for ( var i=0;isSignup(signup) && i < Input (2).actions().size; i++
     ) {                                                              //(2)
        var f = Input(2).actions().get(i);
        var pos = f.getPosition();
        IMPLIES (
             afterLogin( f ) &&                                       //(3)
             CREATE(Input(3), addAction( Input(2), pos+1, signup))    //(4)
             ,
             different(                                               //(5)
                    Output(Input(3), pos).getSession(),
                    Output(Input(3), pos+1).getSession())
     );//end-IMPLIES
    }//end-for
   }//end-for
  }
 }//end-MR
}//end-package
```

(1) The first loop iterates over the inputs to find a sign up action. (2) The second loop 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. (3) Checks if the current action has been performed after a login. (4) Defines a follow-up input with the sign up action being duplicated in a certain position. (5) Checks if the session ID of the response page sent after the two successive login actions is different.

Figure 2.17: OTG_SESS_003: Testing for session fixation

```
1  MR CWE_94_96_B {
2 {
3   keepDialogsOpen = true;                                          //(1)
4   for (Action action : Input(1).actions()){                        //(2)
5      var pos = action.getPosition();                               //(3)
6      for (var x=0; action.containFormInput() && x < action.formInputs.
7      size; x++){                                                   //(4)
8         IMPLIES(
9             action.isClickOnButton &&                             //(5)
10            notTried(x+action.url, Input(1).actions().get(pos).
11            getElementURL()) &&                                   //(6)
12            userCanRetrieveContent(action.getUser(), Output(Input(1),
13            pos)) &&                                              //(7)
14            CREATE(Input(2), Input(1) ) &&                        //(8)
15            Input(2).actions.get(pos).setFormInput(x,
16            StaticInjectionString())                              //(9)
17            ,
18        OR(                                                      //(10)
19            isError(Output(Input(2), pos)),
20            userCanRetrieveContent(action.getUser(), Output(Input(2),pos)))
21     );//end-IMPLIES
22    }//end-for
23   }//end-for
24  }
25 }//end-MR
26 }//end-package
```

(1) It avoids clicking on OK; because dialogs are normally ignored by our framework by clicking on OK. (2) For loop iterate over all actions of the Input(1). (3) Define a variable to store the position of the Input(1)'s action. (4) The second loop iterates over all parameters of the action if the action contains a form input. (5) makes sure the user will submit the form. (6) Verifies the current parameter was not tested before. (7) Filters out the web pages with dynamic content. (8) Creates the follow-up input by copying the Input(1). (9) Injects some Static injection strings to the follow-up input. (10) The system may show an error page to the user or the user is retrieving the content that has right to access it.

Figure 2.18: CWE_94_96_B: Testing for static code injection

```
 MR CWE_792_793_794_795_796_797_A{
{
  keepDialogsOpen = true;                                          //(1)
  for ( Action action : Input(1).actions() ) {                     //(2)
    for ( var par=0; par < action.getParameters().size(); par++ ){ //(3)
       var pos = action.getPosition();                             //(4)
       var value = Input(1).actions().get(pos).getParameterValue(
       par);                                                       //(5)
       IMPLIES(
           notTried( Input(1).actions().get(pos).getUrl() ) &&     //(6)
           value != Boolean &&                                     //(7)
           CREATE(Input(2), Input(1))&&                            //(8)
           Input(2).actions().get(pos).setParameterValue(par,
                SCInjection_beginning(value,SpecialCharacters()))  //(9)
           ,
        OR(                                                        //(10)
           isError(Output(Input(2), pos)),
           EQUAL(Output(Input(1), pos),Output(Input(2), pos)))
     );//end-IMPLIES
     }//end-for
    }//end-for
  }}//end-MR
}//end-package
```

(1) It avoids clicking on OK; because dialogs are normally ignored by our framework by clicking on OK. (2) For loop iterates over all actions of the Input(1). (3) Iterates over all parameters of each action. (4) Stores the parameters of the action in the variable. (5) Reads the user's input value and keeps it in a variable. (6) Checks if the URL was not tried before, to speed up the process. (7) Filters out the boolean input values. (8) Creates the follow-up input by copying the Input(1). (9) Sets the new input value which contains special character for the follow-up input. (10) Verifies that the system should show an error page or it would neutralize the input.

Figure 2.19: CWE_792_793_794_795_796_797_A: Testing for incomplete filtering of one or more instances of special elements

```
MR CWE_20_73_99_219_220_528_530_642_732_OTG_AUTHZ_001a {
{
   for ( Action action : Input(1).actions() ){                     //(1)
      for ( var par=0; par < action.getParameters().size(); par++ ){//(2)
        var pos = action.getPosition();                           //(3)
        IMPLIES(
            notTried( Input(1).actions().get(pos).getUrl() ) &&   //(4)
            CREATE( Input(2), Input(1) )   &&                     //(5)
            Input(2).actions().get(pos).setParameterValue(par,
            RandomFilePath())                                     //(6)
            ,
        OR(                                                       //(7)
            isError(Output(Input(2),pos)),
            userCanRetrieveContent(action.getUser(), Output(Input(2),pos)
            ))
     );//end-IMPLIES
    }//end-for
   }//end-for
}}//end-MR
}//end-package
```

(1) Iterates over the actions of the Input(1). (2) The second for loop iterates over the parameters of the action. (3) Stores the parameters of the current action in a variable. (4) To speed up the process, verifies that the URL has not been tried before. (5) Creates the follow-up input, named Input(2). (6) Sets the value of a parameter to a random file path. (7) Verifies that the system shows an error page or the returned content is something that the user has the right to access.

Figure 2.20: CWE_20_73_99_219_220_528_530_642_732_OTG_AUTHZ_001a: Testing for directory traversal

2.9 Analysis of *MST-wi*'s Applicability and Testability Guidelines

In this section, we investigate (i) the types of security testing activities presenting an oracle problem that can only be addressed by *MST-wi*, (ii) the types of security vulnerabilities that can be identified by *MST-wi*, and (iii) the guidelines (hereafter testability guidelines) that engineers should follow to make *MST-wi* as effective as possible. We investigate the following Research Questions (RQs):

- ***RQ1. To what extent can MST-wi address the oracle problem in the context of security testing?*** This research question aims to identify the security testing activities that, due to the oracle problem, can only be automated using *MST-wi*.
- ***RQ2. What vulnerability types can MST-wi detect?*** *MST-wi* has been designed and implemented to perform security testing by reasoning on outputs of multiple interactions with the system under test. Not every type of vulnerability can be discovered through relationships between outputs of multiple user-system interactions (e.g., some may require program analysis). This research question aims to determine, in a systematic way, the types of security vulnerabilities that *MST-wi* can discover and compare *MST-wi* with state-of-the-art (SOTA) security testing tools.
- ***RQ3. What testability guidelines can we define to enable effective test automation with MST-wi?*** Software testability is the degree to which a software artifact (i.e., a software system, module, requirements, or design document) supports its testing [234]. A higher degree of testability results in decreased test effort, increased quality of test activities, a higher probability of finding software defects, and, as a result, higher-quality software. This question investigates if it is possible to identify testability guidelines that assist engineers in designing, implementing, and configuring their software to enable effective test automation with *MST-wi*.

2.9.1 Targeted Vulnerabilities and Testing Activities

To address our first research question, we study the security testing activities recommended by OWASP [27]. These activities have been described as part of the OWASP testing guidelines Version 4.0 [146] to help engineers understand the *what*, *why*, *when*, *where*, and *how* of testing the security of Web applications. The project provides these activities as a complete testing framework, not merely a simple checklist or prescription of issues that should be addressed. There are, in total, 90 testing activities organized into 11 categories. For instance, in testing activity *Testing Session Timeout*, engineers check that the system under test automatically logs out a user when that user has been idle for a certain amount of time, ensuring that it is not possible to "reuse" the same session and that no sensitive data remains stored in the browser cache [28]. In our analysis, we include the security

testing activities recommended by OWASP because they provide a comprehensive list of security testing methods that we can analyze to identify whether they suffer from the oracle problem.

To address our second and third research questions in a systematic way, we study the list of weaknesses reported in the Common Weakness Enumeration (CWE) database [14]. We provide the following definitions of *vulnerability* and *weakness* since their definitions in the CWE framework [22] lack clarity[2]. A *vulnerability* is a specific fault of the system under test that causes the system to breach 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* [6], *software development concepts* [9], *research concepts* [8], *software fault patterns* [21], *most dangerous errors* [5], and *hardware design* [7]. Other views map the weaknesses to some security-related catalogs (e.g., OWASP Top 10 [11] and SERT CEI C Coding standards [20]).

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 [203, 204]. A weakness is thus the result of a design principle not being followed. For instance, the weaknesses in the design principle *Authenticate Actors* are related to authentication-based components in the system. These components deal with verifying that the actor interacting with the system is who she claims to be. The weaknesses in this category lead to a degradation of the quality of authentication if they are not addressed when designing and implementing the system under test [6]. The views for *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 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 [49] or high-level descriptions of the faulty software behaviour (e.g., incorrect comparison and improper access control).

In our analysis, we focus on the weaknesses in the CWE view for common security architectural tactics [6], the weaknesses in the view for the CWE Top 25 most dangerous software errors (CWE Top 25) [5], and the weaknesses in the view for the OWASP Top 10 Web security risks (OWASP Top 10) [23]. We select the common security architectural

[2]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 [22]'.

Table 2.5: Subset of the security testing activities recommended by OWASP.

Security Testing Category	Security Testing Activity	Oracle Automation Strategy
Configuration and Deployment Management Testing	Test Application Platform Configuration	Manual Oracle
	Test HTTP Methods	No Oracle Needed
Authentication Testing	Testing for Credentials Transported over an Encrypted Channel	Metamorphic Testing
	Testing for Default Credentials	Catalog-based
	Testing for Weak Password Policy	Catalog-based
	Testing for Weak Lock Out Mechanism	Implicit Oracle
Authorization Testing	Testing Directory Traversal File Include	Metamorphic Testing
	Testing for Bypassing Authorization Schema	Metamorphic Testing
	Testing for Privilege Escalation	Metamorphic Testing
	Testing for Insecure Direct Object References	Metamorphic Testing
Session Management Testing	Testing for Cookies Attributes	Manual Oracle
	Testing for Session Fixation	Metamorphic Testing
	Testing for Exposed Session Variables	No Oracle Needed
	Testing for Cross Site Request Forgery	Vulnerability Specific
Input Validation Testing	Testing for SQL Injection	Vulnerability Specific
	Testing for HTTP Verb Tampering	Metamorphic Testing
	Testing for Buffer Overflow	Implicit Oracle
Business Logic Testing	Test for Process Timing	No Oracle Needed
	Test Number of Times a Function Can Be Used Limits	Metamorphic Testing
	Testing for the Circumvention of Work Flows	Manual Oracle

tactics view because it enables us to determine the security design principles that *MST-wi* can verify. We do not consider the view for *software development concepts* because *MST-wi* is a black-box testing approach, that does not aim to discover specific implementation errors (e.g., type errors). We also ignore the views for *software fault patterns*, *research concepts*, and mappings to *coding standards* [20] since they mainly focus on software implementation. We ignore the CWE view for *hardware design* since *MST-wi* does not consider hardware vulnerabilities.

In our analysis, we include the CWE Top 25 and OWASP Top 10 views to assess to what extent *MST-wi* can detect 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 typically *allow attackers to completely take over the control of software, steal data, or prevent software from working [5].* The OWASP Top 10 [11] is the list of the ten most common web application security risks edited by the Open Web Application Security Project [10], 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 includes 43 weaknesses grouped into 10 categories [23].

2.9.2 RQ1: Oracle Problem

Analysis Procedure

To respond to RQ1, we study the security testing activities recommended by OWASP [146]. We systematically analyzed them to identify applicable, SOTA oracle automation strategies. For each activity, we first inspect its description, objective, methods, and tools, if available.

Based on the information collected from our inspection, we identify oracle automation strategies, including metamorphic testing, that can be applied to address the oracle problem if the activity is subject to it. For instance, testing activity *Testing for Bypass Authorization Schema* focuses on verifying how the authorization schema has been implemented for each role or privilege to get access to reserved functions and resources [187]. One of its testing methods is to try to access resources assigned to a different role; another one is to try to access administrative functions. In the first method, it is not always feasible to verify the access to resources with different privileges and roles when the expected outputs need to be identified for a large set of inputs (i.e., the oracle problem). To address the oracle problem in the activity, we specify the MR in Fig. 2.2. It compares the outputs of the executions of the system under test for the same resource and different credentials. We discuss the proportion of software testing activities that cannot be automated with SOTA oracle automation strategies but can be automated with *MST-wi*.

Results

In our analysis, we identified four oracle automation strategies for security testing: *implicit oracle*, *catalog-based*, *vulnerability-specific*, and *metamorphic testing*. In the following, we discuss the details of each strategy; in addition, we discuss why in certain cases *no oracle is needed* or only a *manual oracle* is feasible. To exemplify our descriptions, we provide in Table 2.5, for each OWASP testing category, some of its security testing activities along with oracle automation strategies.

No oracle needed. Some activities collect data to reverse engineer the system under test. They do not verify the security properties of the system. They aim to retrieve information which might be useful to identify potential weaknesses (e.g., the use of vulnerable versions of a Web framework). Therefore, these activities do not have an oracle problem. For instance, in security testing activity *Map Application Architecture* [183], engineers identify the components of a Web system, e.g., reverse proxy, type of front-end Web server, and version of the LDAP server. Commonly, hundreds of Web applications are hosted on an interconnected Web server infrastructure. A single vulnerability in one application may risk the security of the entire infrastructure. Even small risks may evolve into severe ones for other applications on the same server. Therefore, it is important to perform an in-depth review of known security issues for each application. Before the review, engineers need to map the application architecture through some tests to determine which application components are used [183].

Manual oracle. Some activities require humans to determine vulnerabilities based on system specifications. For instance, testing activity *Testing for the Circumvention of Work Flows* concerns vulnerabilities for the misuse of a system in a way that allows malicious users to circumvent the intended workflow [182]. Vulnerabilities related to the circumvention of workflows are very system-specific. In short, the business process of the system under test must ensure that transactions and actions proceed in the right order. For example, when testing a pay-per-view system (i.e., a pay television service by which a viewer can purchase events to view via private telecast), it is necessary to determine transactions that should not enable service access. Only a human can decide, based on system specifications, whether pending transactions should grant service access. Although oracles that present similarities with the exemplifying case described above might be automated in some contexts (e.g., by encoding the logic to decide if a transaction can be performed), their implementation requires substantial manual effort and is unlikely to be generalizable and integrated with test input generation approaches.

Implicit oracle. We can automate some of the security testing activities by following randomized test input generation strategies relying on implicit oracles. An implicit oracle refers to the detection of "obvious" faults such as a program crash [43]. For instance, testing activity *Testing for Buffer Overflow* in Table 2.5 is automated by looking for system crashes in response to lengthy inputs [184]. One of the testing methods in the activity is testing for the format string [24]. A format string is a null-terminated character sequence with conversion specifiers interpreted or converted at run-time. For systems concatenating user input with a format string, we add additional conversion specifiers to cause a buffer overflow and eventually a system crash.

Catalog-based. We can automate some activities based on a predefined catalog in which we specify test inputs and oracles. For instance, testing activity *Testing for Default Credentials* focuses on systems installed on servers with minimal configuration or customization by the server administrator [181]. Often these systems are not properly configured, and the default credentials for initial authentication and configuration are never changed. These default credentials are well-known by malicious users, who use them to access various types of systems. As part of the security testing activity, we use a catalog of default credentials to test whether easy-to-guess pairs of usernames and passwords (e.g., ⟨admin, admin⟩) can be used to log into the system under test.

Vulnerability-specific. Some activities can get automated by SOTA tools such as Burp Suite [193] and thus may not necessarily benefit from MT. These include OWASP testing activities that detect cross-site scripting and code injection vulnerabilities. Other activities are either not targeted or partially automated. For example, Burp Suite does not automate oracles for activity *Testing for Bypassing Authorization Schema* [195]. Though it enables engineers to compare the content of site maps [194] recorded in different user sessions (e.g., with and without certain privileges), it requires engineers to manually identify privileged resources and inspect the differences in the observed system outputs, which is error-prone (e.g., overlooking resources) and expensive. Even Burp Suite plug-ins using Crawljax to build site maps do not address the oracle problem but generate JUnit tests that simply retrieve the mapped resources [125]. With *MST-wi*, engineers, instead, can focus on the specifications of system-level properties without performing such manual testing activities.

Table 2.6: Oracle automation strategies for the OWASP security testing activities*.

Security Testing Category	**Oracle Automation Strategy**					
	Implicit Oracle	**Catalog-based**	**No Oracle Needed**	**Manual Oracle**	**Vulnerability-specific**	***MST-wi***
Information Gathering	-	-	10	-	-	-
Configuration and Deployment Management Testing	-	1	4	3	-	1
Identity Management Testing	-	2	-	3	-	-
Authentication Testing	1	3	-	3	-	3
Authorization Testing	-	-	-	-	-	4
Session Management Testing	-	-	1	2	1	4
Input Validation Testing	1	-	1	2	11	2
Testing for Error Handling	-	-	2	-	-	-
Testing for Weak Cryptography	-	-	-	3	-	1
Business Logic Testing	-	-	1	6	1	1
Client Side Testing	-	-	-	3	9	-
Total	2	6	19	25	22	16
% of testing activities	2%	7%	21%	28%	24%	18%

*Details are available online [55]. For readability, the symbol '-' stands for zero. The *% of testing activities* (last row) is computed with respect to the 90 activities in the OWASP book [146].

Testing activities, including oracles, are automated by the *MST-wi* framework.

MST-wi. In general, *MST-wi* can automate the testing of activities that verify if a resource of the system under test can be accessed under circumstances that should prevent it (e.g., an unauthenticated user or unencrypted channel). These activities benefit from *MST-wi* since they entail the verification of numerous system resources and specific security properties (e.g., each Web page might be accessed by a different set of users). For instance, testing activity *Testing Directory Traversal File Include* focuses on reading directories or files which normally cannot be read, accessing data outside the web document root, and including scripts and other kinds of files from external websites [186]. With a large set of test inputs, it is not feasible to list all the directories and files which a user normally cannot reach. To address the oracle problem in this testing activity, we specify an MR which verifies that a file path should never enable a user to access data that is not provided by the user interface (see relation `CWE_20_..._OTG_AUTHZ_001a` in Fig. 2.20).

Table 2.6 presents a summary of the OWASP security testing categories and the oracle automation strategies. The first column lists the security testing categories. The rest of the columns present the numbers of testing activities in the categories automated by each strategy.

In total, 19 out of 90 activities (21%) do not require a test oracle while 71 activities (79%) do. Also, only 30 out of these 71 activities (42%) can benefit from existing oracle automation solutions (i.e., implicit oracle, catalog-based, and vulnerability-specific), thus highlighting the severe impact of the oracle problem on security testing automation.

More than half of the activities not requiring a test oracle (53%) are associated with the testing category *Information Gathering* because it covers all the activities which require reverse engineering or manual information retrieval to collect data about the system under test.

Among traditional oracle automation solutions, vulnerability-specific approaches are the ones covering the largest proportion of OWASP activities (i.e., 22 out of 90, 24%). Half of these activities are organized into the category *Input Validation Testing* and concern cross-site scripting and injection vulnerabilities; however, as reported in Section 2.9.3, *MST-wi* can still be used to test for these vulnerabilities thus enabling engineers to rely on a single testing framework (i.e., *MST-wi*) rather than several ones. *Catalog-based* and *Implicit* oracles address a low percentage of activities (7% and 2%, respectively). The limited applicability of implicit oracles shows that most of the solutions relying on them (e.g., fuzz testing tools [133]), though useful, partially address the needs of security testing engineers.

Among the activities that cannot benefit from traditional oracle automation solutions, 16 (39%) can be automated thanks to *MST-wi*, which highlights the relevance of the contribution of this study. A majority of these activities (69%) are in the categories *Authentication Testing*, *Authorization Testing*, and *Session Management Testing*. The testing activities in these three categories require multiple interactions between the system under test and one or more users (actors), two characteristics well supported by *MST-wi*. Among them, *MST-wi* can automate all the testing activities in the category *Authorization Testing* (i.e., the activities *Testing Directory Traversal File Include*, *Testing for Bypassing Authorization Schema*, *Testing for Privilege Escalation*, and *Testing for Insecure Direct Object References* in Table 2.5). Further, it can automate half of the activities in the category *Session Management Testing*. An example is *Testing for Session Fixation*, which is automated by `OTG_SESS_003` in Fig. 2.17. A session fixation vulnerability occurs when the system under test does not renew its session cookie(s) after successful user authentication [188]. `OTG_SESS_003` verifies whether a signup action leads to a new session ID, even when the action is performed by a user already logged in.

Based on the above, *we conclude that MST-wi can play a key role in addressing the oracle problem in security testing*. The activities for which *MST-wi* can automate oracles are mostly those that (i) verify that resources can be accessed only by authorized users (*Authorization Testing*), (ii) test the initial authentication and the transfer of the user's authentication data (*Authentication Testing*), (iii) discover vulnerabilities associated with session management (*Session Management Testing*), and (iv) test the system's response to HTTP methods and parameters (*Input Validation Testing*).

2.9.3 RQ2: Vulnerability Types

Analysis Procedure

We aim to study which types of vulnerabilities can be discovered by *MST-wi*. To enable a discussion structured according to well-defined categories of vulnerabilities, we compute, for each category in the CWE views mentioned in Section 2.9.1 (i.e., *Common security*

Table 2.7: Subset of the security weaknesses in the CWE view for common security design principles.

Design principle	Weakness	Belongs to Top 25	Belongs to Top 10	Generic Weakness	Addressed by *MST-wi*	Testability Feature (TF) / Reason *MST-wi* cannot be applied (R)
Audit	Omission of Security-relevant Information	No	Yes	Yes	No	R3
	Obscured Security-relevant Information by Alternate Name	No	No	Yes	No	R3
Authenticate Actors	Improper Authentication	Yes	Yes	Yes	Yes	TF3
	Weak Password Recovery Mechanism for Forgotten Password	No	Yes	Yes	No	R2
Authorize Actors	Improper Privilege Management	No	Yes	Yes	Yes	TF3
	Process Control	No	No	Yes	No	R1
Encrypt Data	Small Space of Random Values	No	No	Yes	No	R4
	Missing Encryption of Sensitive Data	No	Yes	Yes	No	R3
Identify Actors	Improper Validation of Certificate with Host Mismatch	No	No	Yes	No	R5
	Improper Validation of Certificate Expiration	No	No	Yes	Yes	TF10
Limit Access	Improper Restriction of XML External Entity Reference	Yes	Yes	Yes	Yes	TF2
	External Control of File Name or Path	No	Yes	Yes	Yes	TF1
Manage User Sessions	J2EE Bad Practices: Non-serializable Object Stored in Session	No	Yes	No	No	R1
	Insufficient Session Expiration	No	Yes	No	Yes	TF2, TF8
Validate Inputs	Cross-site Scripting	Yes	Yes	Yes	Yes	TF2
	Deserialization of Untrusted Data	Yes	Yes	No	No	R2

architectural tactics, *CWE Top 25*, and *OWASP Top 10*), the percentage of weaknesses that can be automatically discovered by *MST-wi*.

We systematically analyzed all the weaknesses with the objective of writing, for each one, one or more MRs using SMRL. For each weakness, we first inspect its description, its demonstrative examples, the description of concrete vulnerabilities (CVE) and common attack patterns (CAPEC) [13] associated with the weakness. Based on the information collected from our inspection, we implemented, using SMRL, a new MR that targets the weakness or reused, if possible, an MR already available in the *MST-wi* catalog. Each time we could not do so, we kept track of the reasons preventing the writing of an MR. All the MRs resulting from this analysis are part of the MRs catalog provided online [57].

We report the percentage of weaknesses for which it has been possible to implement an MR; in other words, the weaknesses that can be automatically tested with *MST-wi*. Since some of the weaknesses in the CWE database are specific to certain types of systems (e.g., Java Enterprise [25]), we distinguish between the results achieved with all the weaknesses (i.e., generic and specific), and the results achieved with the generic weaknesses only.

To better characterize the weaknesses that cannot be identified by *MST-wi*, we analyze the distribution of the reasons preventing its application, 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 lists. To this end, we rely on the definitions available on the CWE Web site. More precisely, we rely on the list of CWE weakness IDs

Table 2.8: Summary of the CWE architectural security design principles and weaknesses targeted by *MST-wi*.

Security Design Principle	Weaknesses		Targeted weaknesses	
	all	generic	all	generic
Audit	6	6	1 (16%) 10^{th}	1 (16%) 10^{th}
Authenticate Actors	28	27	12 (43%) 4^{th}	12 (44%) 4^{th}
Authorize Actors	60	55	34 (57%) 3^{rd}	34 (62%) 3^{rd}
Cross Cutting	9	8	3 (33%) 6^{th}	3 (37%) 6^{th}
Encrypt Data	38	37	8 (21%) 8^{th}	8 (22%) 8^{th}
Identify Actors	12	12	3 (25%) 7^{th}	3 (25%) 7^{th}
Limit Access	8	5	3 (38%) 5^{th}	2 (40%) 5^{th}
Limit Exposure	6	6	0 (0%) 11^{th}	0 (0%) 11^{th}
Lock Computer	1	1	0 (0%) 11^{th}	0 (0%) 11^{th}
Manage User Sessions	6	4	4 (67%) 2^{nd}	4 (100%) 1^{st}
Validate Inputs	39	33	31 (79%) 1^{st}	29 (88%) 2^{nd}
Verify Message Integrity	10	10	2 (20%) 9^{th}	2 (20%) 9^{th}
Total	223	204	101 (45%)	98 (48%)

belonging to the views for CWE Top 25 and OWASP Top 10 provided on the CWE Web site. Within these lists, we identify the CWE IDs belonging to the CWE view for common security architectural tactics, our main target in this study, and track the CWE IDs tested by our MRs.

To provide concrete examples of the weaknesses in our analysis, we report, in Table 2.7, a subset of the weaknesses in the CWE view for common security architectural tactics. We refer to Table 2.7 in the rest of the section. Columns *Design principle* and *Weakness* report the security design principle affected by a weakness and its name, respectively. Columns *Belongs to Top 25* and *Belongs to Top 10* indicate whether a weakness also belongs to the *CWE Top 25* or the *OWASP Top 10* view, respectively. We also indicate if the weakness can be targeted by *MST-wi*. The remaining columns refer to concepts introduced later in this section.

Finally, to compare the capabilities of *MST-wi* with SOTA approaches, we refer to a recent empirical study from Elder et al. [81] comparing SOTA vulnerability detection tools based on dynamic and static program analysis. In the security context, static and dynamic program analysis tools are often referred to as Static application security testing (SAST) and Dynamic Application Security Testing (DAST). Elder et al. consider two open-source tools (i.e., OWASP Zap [185] and Sonarqube [217] for DAST and SAST, respectively) and another two proprietary tools whose name has not been disclosed due to licensing contracts (they are referred to as *Dynamic Analysis 2 - DA2* and *Static Analysis 2 - SA2)*. To the best of our knowledge, the study of Elder et al. is the only one providing the list of CWE IDs targeted by SOTA security tools (see Table 10 in their paper). Based on their list, we determine which CWE IDs belonging to the sets considered in our study are targeted by Zap, Sonarqube, SA2, and DA2. Using the collected data, we compare *MST-wi* with SOTA static and dynamic analysis regarding the overall number of targeted vulnerabilities. Also, we determine the number of vulnerability types that can be discovered only by *MST-wi* and not by a single competing approach or by any of the four approaches. To be adopted in practice, *MST-wi* should complement SOTA tools (i.e., target weaknesses not considered by these tools).

Table 2.9: Summary of the CWE Top 25 weaknesses targeted by *MST-wi*.

Weaknesses		Targeted weaknesses	
all	**generic**	**all**	**generic**
25	18	15 (60%)	14 (78%)

Table 2.10: Summary of the security weaknesses for OWASP Top 10 security risks targeted by *MST-wi*.

OWASP Security Risk	**Weaknesses**		**Targeted weaknesses**	
	all	**generic**	**all**	**generic**
Broken Access Control	20	18	15 (75%)	15 (83%)
Cryptographic Failures	24	24	3 (13%)	3(13%)
Injection	21	18	18 (86%)	16 (89%)
Insecure Design	22	20	12(55%)	12 (60%)
Security Misconfiguration	5	4	3 (60%)	2 (50%)
Vulnerable and Outdated Component	0	0	0 (0%)	0 (0%)
Identification and Authentication Failures	20	20	11 (55%)	11 (55%)
Software and Data Integrity Failures	9	8	1 (11%)	1 (13%)
Security Logging and Monitoring Failures	4	4	1 (25%)	1 (25%)
Server-Side Request Forgery (SSRF)	1	1	0 (0%)	0 (0%)
Total	126	117	64 (51%)	61 (52%)

Results

Table 2.8 presents a summary of the CWE security design principles and related security weaknesses targeted by *MST-wi*. The first column in Table 2.8 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 weaknesses that can be automatically discovered by *MST-wi* among all and generic weaknesses (we report the number, percentage, and ranking).

In total, 101 out of all 223 weaknesses (45%) and 98 out of 204 generic weaknesses (48%) in the view can be discovered by *MST-wi*. These numbers show that our approach enables engineers to automatically discover a large subset of the weaknesses. Readers can download the details of our analysis for all 223 weaknesses from our replicability package [57]. If we sort security design principles based on the percentage of weaknesses targeted by *MST-wi*, we observe that the rankings for generic and all weaknesses match except for the first two security design principles in the rankings, which are swapped. These top ranked security design principles are *Manage User Sessions* (1^{st} for generic weaknesses, 2^{nd} considering all the weaknesses) and *Validate Inputs* (2^{nd} for generic weaknesses, 1^{st} considering all the weaknesses). They are about malicious actors accessing resources because of session management faults (*Manage User Sessions*), and malicious actors providing malformed input data (e.g., code injection) to the system (*Validate Inputs*). The main reason for the difference is that specific weaknesses for session management faults include two weakness

Table 2.11: Reasons preventing the application of *MST-wi*.

ID	Reason
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	The weakness can be discovered only by controlling a third-party component.

Table 2.12: Distribution of reasons preventing the application of *MST-wi* to verify security design principles. The second column (#) reports the number of weaknesses not discovered by *MST-wi*.

Security Design Principle	#	R1	R2	R3	R4	R5	Sum
Audit	5		2	3			5
Authenticate Actor	16		11	3	1	1	16
Authorize Actor	26	15	6	3	1	1	26
Cross Cutting	6	1	1	4			6
Encrypt Data	30	1	21	2	6		30
Identify Actors	9	1	2	3		3	9
Limit Access	5	1	1	3			5
Limit Exposure	6	1	3	2			6
Lock Computer	1			1			1
Manage User Sessions	2	1		1			2
Validate Inputs	8	1	7				8
Verify Message Integrity	8		6			2	8
Total	122	22	**60**	25	8	7	122

(i.e., CWE-6 and CWE-579) related to the serialization of J2EE objects that cannot be discovered through MT but only through code inspection. Indeed, CWE-6 is discovered by determining if the session includes a nonserializable object; CWE-579 can be discovered by checking if the session ID is stored in a field that is shorter than the one used in the J2EE session object. Since the rest of the rankings match, in the following discussion, we report only the percentages for generic weaknesses to simplify reading.

Other security design principles with a high percentage of weaknesses (above 40%) being detected by *MST-wi* are *Authorize Actors* and *Authenticate Actors* which concern malicious actors (external systems or users) accessing resources they are not authorized to access. These weaknesses are often discovered through interactions with the system and can be tested with *MST-wi*. *MST-wi* cannot cover cases in which program analysis is needed (e.g., *Insufficient Compartmentalization* and *Reliance on Security Through Obscurity*).

MST-wi targets a low percentage (below 20%) of the weaknesses related to the security design principles *Audit*, *Limit Exposure*, and *Lock Computer* (i.e., 16%, 0%, and 0% respectively). Such percentages are due to *MST-wi* relying, for source inputs, on sequences of user-system interactions. The weaknesses related to *Audit*, *Limit Exposure*, and *Lock Computer* are, on the contrary, about quality of recorded logs, information that the system exposes, and restrictions of the lockout mechanism (e.g., lock an account after a predefined number of failed logins). They all require manual data inspection.

Unsurprisingly, the design principles *Authorize Actors* (34 weaknesses), *Validate Inputs* (31), and *Authenticate Actors* (12) also have a high number of weaknesses targeted by *MST-wi*. Indeed, they concern interactions between external actors and the system, which is the main focus of *MST-wi*.

Table 2.9 gives a summary of the CWE Top 25 weaknesses targeted by *MST-wi*. It can automatically discover 15 out of the 25 CWE top weaknesses (60%) and 14 out of the 18 generic CWE top weaknesses (78%), which shows that *MST-wi* is a key solution to identify widely spread weaknesses.

Among the CWE Top 25 weaknesses, *MST-wi* cannot detect the ones that require program analysis or interactions with third parties (e.g., other system users or system administrators) to be detected: *Out-of-bounds Write*, *Out-of-bounds Read*, *Improper Neutralization of Special Elements used in an OS Command*, *Use After Free*, *Integer Overflow or Wraparound*, *Deserialization of Untrusted Data*, *NULL Pointer Dereference*, *Use of Hard-coded Credentials*, *Improper Restriction of Operations within the Bounds of a Memory Buffer*, and *Server-Side Request Forgery (SSRF)*.

Table 2.10 presents a summary of the security weaknesses related to the OWASP Top 10 security risks targeted by *MST-wi*. It targets 64 out of the 126 weaknesses (51%) and 61 out of the 117 generic weaknesses (52%) in this view. It targets a high percentage (above 55%) of the weaknesses leading to security risks *Broken Access Control*, *Injection*, *Insecure Design*, *Security Misconfiguration*, and *Identification and Authentication Failures* (i.e., 75%, 86%, 55%, 60%, and 55%, respectively). These risks are about unauthorized access to resources, injecting malicious client-side scripts into a website, leveraging the lack of security controls (e.g., an unprotected primary channel), gaining system information thanks to system misconfiguration, and bypassing authentication. All involve malicious user-system interactions.

MST-wi can target none of the weaknesses related to *Vulnerable and Outdated Components*, which concern the use of outdated libraries affected by known vulnerabilities. Although it would be feasible to implement MRs detecting failures of specific components, we cannot provide a catalog of MRs that cover all the outdated libraries in the market. Therefore, we excluded them from our analysis.

Table 2.11 presents the reasons preventing the application of *MST-wi* to discover weaknesses. *MST-wi* has been designed to select source inputs by relying either on a Web crawler or on manually implemented test scripts that automate the user-system interactions. Source inputs are automatically turned into follow-up inputs which are used to discover vulnerabilities. *MST-wi* can discover vulnerabilities that can be exercised through a sequence of interactions. Therefore, *MST-wi* cannot be applied when the weakness concerns a system that is not Web-based, or when it concerns components of a Web system that exchange data through dedicated protocols (see R1 in Table 2.11). Also, some weaknesses can be discovered only through program analysis, which *MST-wi* does not support (see R2 in Table 2.11). 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 [41]. Example cases are reported in Table 2.7 and concern log files omitting or

Table 2.13: Distribution of reasons preventing the application of *MST-wi* to discover weaknesses associated with the OWASP Top 10 security risks. The second column (#) reports the number of weaknesses not discovered by *MST-wi*.

OWASP Security Risk	**#**	**R1**	**R2**	**R3**	**R4**	**R5**	**Sum**
Broken Access Control	5	1	3	1			5
Cryptographic Failures	21		18	1	2		21
Injection	3	1	2				3
Insecure Design	10	2	6	2			10
Security Misconfiguration	2		2				2
Vulnerable and Outdated Component	0						0
Identification and Authentication Failures	9	1	4	1		3	9
Software and Data Integrity Failures	8	1	6	1			8
Security Logging and Monitoring	3		3				3
Server-Side Request Forgery (SSRF)	1					1	1
Total	62	6	**44**	6	2	4	62

inappropriately reporting information about attempted attacks, relying on a weak password recovery mechanism, improper validation of certificates, configuring a small space for random variables, lack of encryption, storing objects in session tokens, and deserializing untrusted data.

It is not possible to define an MR in cases where a human is needed to inspect the system output (R3). For instance, to discover weakness *Weak Password Recovery Mechanism for Forgotten Password* in Table 2.7, 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 answer can be easily determined [18]).

Due to the MT foundations (based on MRs), *MST-wi* cannot be applied for weaknesses that can be discovered only through data analysis (see R4 in Table 2.11). Some weaknesses can be determined only by analyzing large amounts of data (e.g., log files) based on statistics or machine learning; this analysis cannot be performed with an MR.

Finally, some weaknesses can be discovered only by controlling a third-party system (see R5 in Table 2.11). This is the case for *Improper Validation of Certificate with Host Mismatch* (see Table 2.5), which requires setting up a malicious host with the same IP of the SUT. Expressing such interactions in MRs, in general, is infeasible.

Table 2.12 presents the distribution of reasons preventing the application of *MST-wi* for the weaknesses regarding common security architectural tactics. In 60 out of 122 weaknesses (49%) that cannot be discovered by *MST-wi*, program analysis is required (see R2). This is expected since program analysis complements software testing in software verification. R2 is particularly prevalent for the security design principle *Encrypt Data*, where 21 out of 30 weaknesses (70%) are not addressed due to R2. These 21 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., hash

Table 2.14: Distribution of reasons preventing the application of *MST-wi* to discover CWE Top 25 weaknesses.

Weakness	**R1**	**R2**	**R3**	**R4**	**R5**
Out-of-bounds Write		1			
Out-of-bounds Read		1			
Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	1				
Use After Free		1			
Integer Overflow or Wraparound			1		
Deserialization of Untrusted Data		1			
NULL Pointer Dereference		1			
Use of Hard-coded Credentials			1		
Improper Restriction of Operations within the Bounds of a Memory Buffer		1			
Server-Side Request Forgery (SSRF)					1
Total	1	**6**	2	0	1

without a salt). Therefore, in these cases, a static program analysis approach should be used; for example, to find hard-coded cryptography keys.

The second most frequent category is R3, which, in most of the cases, concerns the output generated in an exceptional situation (e.g., messages provided in log files). R3 uniformly affects, with one to four cases, all the design principles except the two principles concerning the handling of input data (*Validate Inputs* and *Verify Message Integrity*). For these two cases, static program analysis is more effective than testing; this is the case for *CWE-391* (*Unchecked Error Condition*), where it is enough to examine source code for missing operations following the verification of function results.

The third most frequent category is R1; however, it is the prevalent reason for not applying *MST-wi* in the case of authorization weaknesses (15 out of 26 weaknesses) and has limited impact on all the other principles (i.e., one or no cases). Authorization is a generic security property that goes beyond Web-based systems; therefore, 15 out of these 26 weaknesses (58%) concern functionalities that are not implemented by Web-systems. They concern the file system (e.g., preserving or managing the permissions related to files), the process control in an operating system, or the process communication in a mobile operating system. These weaknesses cannot be discovered by an automated test framework dedicated to Web-based interactions.

R4 has a limited impact on almost all the design principles except for *Encrypt Data*, where it prevents *MST-wi* from detecting six weaknesses; indeed, in several cases, the limitations of encryption algorithms (e.g., *Insufficient entropy*) can be discovered only through a statistical test.

R5 is the least frequent reason for not applying *MST-wi* because most of the weaknesses are not due to interactions among multiple components.

Tables 2.13 and 2.14 report the reasons preventing the application of *MST-wi* to discover some highly critical vulnerabilities. In these cases as well, the main reason is the necessity to rely on static program analysis. This is expected since testing and program analysis are complementary quality assurance activities.

Table 2.15: Testability features and factors for *MST-wi*.

ID	Testability feature	Testability factor
TF1	The feature under test is accessible via a URL/path	Controllability
TF2	The testing framework supports modifying parameter values	Test support environment
TF3	It is possible to log-in with a predefined list of credentials	Controllability
TF4	System settings or configuration elements can be controlled by the test engineer	Controllability
TF5	The testing framework can control the Web-browser (e.g., click on back button)	Test support environment
TF6	The type of the parameters of the request (in URL or post-data) is known or can be easily determined	Controllability
TF7	It is possible to access system artefacts (e.g., log files)	Observability
TF8	The system under test provides a feature to configure the system time	Controllability
TF9	The testing framework supports handling multiple user sessions in parallel	Test support environment
TF10	The testing framework has a feature to select certificates	Test support environment

Tables 2.16 to 2.18 provide the results of the comparison of *MST-wi* with state-of-the-art SAST and DAST tools. Concerning the security design principle verification (Table 2.16), we conclude that *MST-wi* targets most weaknesses and outperforms the best competing approach (SA2). Also, the set of weaknesses targeted by *MST-wi* is larger than what can be targeted by applying all four competing approaches together. Indeed, all the other approaches can discover only 84 weaknesses (see Column *Weaknesses targeted by any other* in Table 2.16), while *MST-wi* targets 101 weaknesses (see Column *Weaknesses targeted by* MST-wi). Concerning complementarity, we observe that *MST-wi* can detect 56 weaknesses that any other approach cannot target (see column *Weaknesses targeted by MST but not by Any other*). Most of the weaknesses targeted by only *MST-wi* belong to the principles Authorize actors (25), Validate inputs (15), and Authenticate Actors (7), which are the principles with most of the weaknesses targeted by *MST-wi* (see Column *Weaknesses targeted by* MST-wi). Together, all SOTA approaches can target only 39 weaknesses that *MST-wi* does not target (see column *Weaknesses not targeted by MST but targeted by Any other*). The tool targeting most of the weaknesses not targeted by *MST-wi* is SA2, which fares the best in the study of Elder et al. [81]. The design principles in which SOTA tools provide greater benefits (i.e., the number of weaknesses targeted only by *MST-wi* is lower than the number of weaknesses targeted by other tools only) are (1) *Encrypt data*, which we already reported above as one of the weakest for *MST-wi*, (2) *Identify Actors* since SAST tools leverage code inspection to determine lack of certificate validation, (3) *Audit*, which we already reported as benefiting from static analysis (e.g., it can determine lack of logging after exceptions or data-flows from sources of private data to log files), (4) *Limit Access*, where SA2 and DA2 can detect information exposure through messages, (5) *Limit*

Table 2.16: Comparison of *MST-wi* with SOTA approaches for the verification of security design principles. Bold font is used to indicate the approach that performs best for a specific design principle. For the subtable *Weaknesses targeted by*, we highlight the highest value on a row. Precisely, we highlight the highest value in the leftmost subtable, and we compare all columns across the two subtables to the right and highlight the highest value. In the leftmost subtable, we underline cases where *MST-wi* performs better than the joint use of SOTA approaches. Also, in the middle subtable, we underline cases where the number of weaknesses targeted exclusively by *MST-wi* is higher than the number of weaknesses targeted exclusively by the combined use of other approaches.

Security Design Principle	**Weaknesses targeted by**						**Weaknesses targeted by MST but not by**					**Weaknesses not targeted by MST but targeted by**				
	MST	Any other	Zap	DA2	Sonar	SA2	Any other	Zap	DA2	Sonar	SA2	Any other	Zap	DA2	Sonar	SA2
Audit	1	3	0	0	0	**3**	0	1	1	1	0	**2**	0	0	0	**2**
Authenticate Actors	**12**	10	0	2	1	9	**7**	12	11	11	**7**	5	0	1	0	4
Authorize Actors	**34**	13	2	0	1	13	25	32	**34**	**34**	25	4	0	0	1	4
Cross Cutting	**3**	2	0	0	2	0	2	**3**	**3**	2	**3**	1	0	0	1	0
Encrypt Data	8	**18**	2	5	8	**10**	3	8	8	7	4	**13**	2	5	7	6
Identify Actors	3	**7**	1	1	1	**7**	1	3	3	3	1	**5**	1	1	1	**5**
Limit access	3	**5**	0	1	1	**5**	0	**3**	**3**	2	0	2	0	1	0	2
Limit exposure	0	1	1	0	0	1	0	0	0	0	0	**1**	**1**	0	0	**1**
Lock computer	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Manage User Session	**4**	2	0	0	0	2	2	**4**	**4**	**4**	2	0	0	0	0	0
Validate Inputs	**31**	20	10	7	2	14	15	24	25	**30**	19	4	3	1	1	2
Verify Message Integrity	2	**3**	1	0	0	**3**	1	**2**	**2**	**2**	1	**2**	1	0	0	**2**
TOTAL	**101**	84	17	16	16	67	56	92	94	**96**	**62**	39	8	9	11	28

Table 2.17: Comparison of *MST-wi* with SOTA approaches for the verification of security design principles in the OWASP Top 10 list. Highlight and underlining follow the same conventions as Table 2.16.

OWASP Security Risk	**Weaknesses targeted by**						**Weaknesses targeted by MST but not by**					**Weaknesses not targeted by MST but targeted by**				
	MST	**Any other**	**Zap**	**DA2**	**Sonar**	**SA2**	**Any other**	**Zap**	**DA2**	**Sonar**	**SA2**	**Any other**	**Zap**	**DA2**	**Sonar**	**SA2**
Broken Access Control	**15**	11	4	3	1	11	5	11	13	**14**	5	1	0	1	0	1
Cryptographic Failures	3	**14**	1	6	6	8	2	3	3	3	2	**13**	1	6	6	**7**
Injection	**18**	16	7	5	0	11	5	14	14	**18**	8	3	3	1	0	1
Insecure Design	**12**	8	3	1	2	4	6	10	**12**	10	9	2	1	1	0	1
Security Misconfiguration	3	3	0	0	1	3	1	**3**	**3**	2	1	1	0	0	0	1
Vulnerable and Outdated Component	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Identification and Authentication Failures	11	**12**	0	2	2	**12**	4	**11**	10	10	4	5	0	1	1	**5**
Software and Data Integrity Failures	1	**3**	2	0	1	**3**	1	1	1	1	1	**3**	2	0	1	**3**
Security Logging and Monitoring Failures	1	**2**	0	0	0	**2**	0	**1**	**1**	**1**	0	**1**	0	0	0	**1**
Server-side Request Forgery (SSRF)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TOTAL	64	**69**	17	17	13	54	24	54	57	**59**	30	29	7	10	8	20

Table 2.18: Comparison of *MST-wi* with SOTA approaches for the verification of weaknesses in the CWE Top 25 list. Highlights follow the same convention as Table 2.16.

Weaknesses targeted by						**Weaknesses targeted by MST but not by**					**Weaknesses not targeted by MST but targeted by**				
MST	**Any other**	**Zap**	**DA2**	**Sonar**	**SA2**	**Any other**	**Zap**	**DA2**	**Sonar**	**SA2**	**Any other**	**Zap**	**DA2**	**Sonar**	**SA2**
15	**15**	6	6	4	14	5	10	10	**12**	6	5	1	1	1	5

Exposure, where ZAP and likely SA2 can be used to detect cross-domain JavaScript file inclusion, and (6) *Verify Message Integrity*, where SA2 and ZAP can be used to check the integrity of cookies and whether there are error conditions not being handled (the latter is enabled only by SA2).

Concerning the security design weaknesses in the OWASP Top 10 (see Table 2.16), we note that *MST-wi* targets most of the weaknesses. *MST-wi* discovers 64 weaknesses, while the best SOTA approach (i.e., SA2) finds 54 weaknesses (see column *Weaknesses targeted by SA2*). We conclude that *MST-wi* complements the four other approaches by detecting 24 weaknesses they do not discover. The best-competing approach (SA2) can target 20 weaknesses not discovered by *MST-wi*, while *MST-wi* targets 30 weaknesses not targeted by SA2. However, all the SOTA approaches detect together 29 weaknesses not targeted by *MST-wi*, which indicates complementarity between *MST-wi* and other tools. SOTA approaches work better for cryptographic failures (13 weaknesses not discovered by *MST-wi* but discovered by others and two weaknesses detected only by *MST-wi*). Such a result is expected since most of those issues are related to outdated encryption algorithms or the wrong setup of security keys, which can be easily detected using program analysis (in particular static analysis). However, *MST-wi* outperforms the other approaches for *Broken Access Control* (indeed, *MST-wi* includes several MRs to determine if resources not available from the GUI can be accessed by modifying inputs), *Code injection* (*MST-wi* includes several MRs that modify requests by relying on catalogs of injections, including special characters, which are not covered by other approaches), and *Insecure design* (similarly to the case of *Broken Access Control*, the detection of failures in the handling of privileges is enabled by *MST-wi*'s MRs verifying if resources not available from the GUI can be accessed by modifying inputs).

As for the CWE Top 25 list (see Table 2.18), the number of weaknesses targeted by *MST-wi* and the best SOTA approach is similar (15 versus 14); however, they complement each other. *MST-wi* targets five weaknesses not targeted by other approaches: *CWE-306 (Missing Authentication for Critical Function)*, *CWE-276 (Incorrect Default Permissions)*, *CWE-732 (Incorrect Permission Assignment for Critical Resource)*, *CWE-77 (Improper Neutralization of Special Elements used in a Command - Command Injection)*, *CWE-434 (Unrestricted Upload of File with Dangerous Type)*. In line with our discussion above, the first four weaknesses concern permission management or command injections not targeted by other approaches; the latter requires test automation not supported by the four SOTA approaches. Indeed, they do not generate test inputs automatically but rely on modifying inputs provided by the end user. *MST-wi*, instead, derives specific inputs for CWE-434. For the weaknesses not targeted by *MST-wi* but by SOTA approaches, we report that dynamic program analysis approaches (Zap and DA2) and SA2 discover a code injection vulnerability (*CWE 78 - OS Command Injection*) that *MST-wi* does not target because it is not specific to Web systems. As discussed above, the other weaknesses not covered by *MST-wi* are due to *them being only discoverable by means of program analysis* (R2 in Table 2.11): *CWE-190 (Integer Overflow or Wraparound)*, *CWE-502 (Deserialization of Untrusted Data)*, *CWE-476 NULL Pointer Dereference*, *CWE-798 Use of Hard-coded Credentials.*

Summary. As a result of our analysis, we conclude that *MST-wi* can target a large

percentage (45%) of the weaknesses organized in the CWE view for common security architectural tactics. Such percentage increases to 48% if we consider generic weaknesses only. *MST-wi* can also target a majority of high-risk weaknesses (51% of the weaknesses related to the OWASP Top 10 security risks and 60% of the CWE Top 25 weaknesses). These results are promising as they demonstrate that *MST-wi* is relevant for a large subset of vulnerabilities occurring in practice. The weaknesses that *MST-wi* cannot target are mostly those (i) that can only be discovered using program analysis, (ii) that are not related to user-system interactions, or (iii) that concern non-Web-based systems. Further, we demonstrated that, for all the category of weakness considered (i.e., security design principles, OWASP Top 10, or CWE Top 25), *MST-wi* complements state-of-the-art SAST and DAST approaches in terms of weaknesses targeted. Specifically, combining *MST-wi* with SA2 seems to be a particularly effective combination as it enables detecting 129 weaknesses (i.e., 101 + 28), that is 92% of the 140 weaknesses that can be detected by any approach. Such complementarity is a key factor that justifies the adoption of metamorphic testing as a solution to improve software security. The weaknesses not discovered by any approach considered in our assessment (83 out of 223) can mainly be discovered through manual code inspection activities (e.g., to determine if *a client/server product performs authentication within client code but not in server code*, CWE-603), activities which are difficult to automate through generic tools since they often require a specific understanding of the system under analysis.

2.9.4 RQ3: Testability guidelines

Analysis Procedure

This research question investigates the possibility to define testability guidelines that support engineers in automatically testing software systems with *MST-wi*. 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 *MST-wi*, in the CWE view for common security architectural tactics. We identify, for each weakness, a set of features necessary to enable automated testing with our approach. For each Web-specific function used in our MRs, we determine the inputs sent to the SUT and the outputs retrieved from the SUT. Further, we determine what SUT's interfaces should receive the inputs selected by the MR and produce the outputs retrieved by the MR; based on them, we derive our testability features. We expect our set of features to be complete because we developed *MST-wi*'s Web-specific functions.

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 [88] lists 21 testability factors: *observability*, *controllability*, *complexity*, *cohesion*, *understandability*, *inheritance*, *reliability*, *availability*, *flexibility*, *test suite reusability*, *maintainability*, *unit*

Table 2.19: Distribution of testability features for *MST-wi*.

Security Design Principle	**Testability Feature**									
	TF1	**TF2**	**TF3**	**TF4**	**TF5**	**TF6**	**TF7**	**TF8**	**TF9**	**TF10**
Audit	-	-	-	-	-	-	1	-	-	-
Authenticate Actors	2	3	2	3	-	-	-	2	-	-
Authorize Actors	14	4	9	2	1	2	1	-	-	1
Cross Cutting	-	3	-	-	-	-	-	-	-	-
Encrypt Data	-	1	1	3	-	-	2	1	-	-
Identify Actors	1	-	-	-	-	-	-	-	-	2
Limit Access	1	1	-	-	-	-	-	-	1	-
Limit Exposure	-	-	-	-	-	-	-	-	-	-
Lock Computer	-	-	-	-	-	-	-	-	-	-
Manage User Sessions	-	1	-	-	1	-	-	1	2	-
Validate Inputs	4	27	-	-	-	-	-	-	-	-
Verify Message Integrity	1	1	-	-	-	-	-	-	-	-
Total	23	41	12	8	2	2	4	4	3	3
% of weaknesses *	22%	40%	12%	8%	2%	2%	4%	4%	3%	3%

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

Table 2.20: Distribution of testability features of *MST-wi* for the weaknesses associated with the OWASP Top 10 security risks.

OWASP Security Risk	**Testability Feature**									
	TF1	**TF2**	**TF3**	**TF4**	**TF5**	**TF6**	**TF7**	**TF8**	**TF9**	**TF10**
Broken Access Control	7	3	4	-	-	-	1	-	-	-
Cryptographic Failures	-	-	-	2	-	-	-	1	-	-
Injection	3	14	-	-	-	-	-	-	1	-
Insecure Design	2	3	3	-	1	1	2	-	-	-
Security Misconfiguration	-	1	-	2	-	-	-	-	-	-
Vulnerable and Outdated Component	-	-	-	-	-	-	-	-	-	-
Identification and Authentication Failures	1	3	2	4	-	-	-	1	1	-
Software and Data Integrity Failures	-	1	-	-	-	-	-	-	-	-
Security Logging and Monitoring Failures	-	-	-	-	-	-	1	-	-	-
Server-side Request Forgery (SSRF) & Monitoring	-	-	-	-	-	-	-	-	-	-
Total	13	25	9	8	1	1	4	2	2	0
% of weaknesses *	20%	38%	14%	12%	2%	2%	6%	3%	3%	0%

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

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 *MST-wi*, 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.

Table 2.21: Distribution of testability features of *MST-wi* for the CWE Top 25 weaknesses.

CWE Top 25 Weakness	**Testability Feature**									
	TF1	**TF2**	**TF3**	**TF4**	**TF5**	**TF6**	**TF7**	**TF8**	**TF9**	**TF10**
Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	-	1	-	-	-	-	-	-	-	-
Improper Input Validation	-	1	-	-	-	-	-	-	-	-
Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	-	1	-	-	-	-	-	-	-	-
Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	1	-	-	-	-	-	-	-	-	-
Cross-Site Request Forgery (CSRF)	-	1	-	-	-	-	-	-	-	-
Unrestricted Upload of File with Dangerous Type	-	-	-	-	-	1	-	-	-	-
Missing Authentication for Critical Function	-	-	1	-	-	-	-	-	-	-
Improper Authentication	-	-	1	-	-	-	-	-	-	-
Missing Authorization	-	-	1	-	-	-	-	-	-	-
Incorrect Default Permissions	-	-	1	-	-	-	-	-	-	-
Exposure of Sensitive Information to an Unauthorized Actor	1	-	-	-	-	-	-	-	-	-
Insufficiently Protected Credentials	-	-	1	-	-	-	-	-	-	-
Incorrect Permission Assignment for Critical Resource	1	-	-	-	-	-	-	-	-	-
Improper Restriction of XML External Entity Reference	-	1	-	-	-	-	-	-	-	-
Improper Neutralization of Special Elements used in a Command ('Command Injection')	-	1	-	-	-	-	-	-	-	-
Total	3	6	5	0	0	1	0	0	0	0
% of weaknesses *	20%	40%	33%	0%	0%	7%	0%	0%	0%	0%

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

Results

Table 2.15 presents the testability features for *MST-wi* and the corresponding testability factors. In total, we have 10 testability features. Six features concern the system under test, while the other four features concern the test environment (see testability factor *Test support environment*).

In the case of *MST-wi*, 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 [88]), (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 [88]), 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 *MST-wi*: security vulnerability testing at the system level by mimicking the actions performed by a malicious user. Therefore, to determine if the output of the system is correct, *MST-wi* 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 2.7. 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 [15]. *MST-wi* can be applied to identify this weakness when the feature under test is accessible through a URL/path (see TF1 in Table 2.15). We match this testability feature to testability factor *Controllability*. In weakness *Insufficient Session Expiration* in Table 2.7, a Web system permits malicious users to reuse old session credentials or session IDs for authorization [17]. *MST-wi* automatically identifies this weakness only when it is possible to modify the values of the parameters passed in HTTP requests (TF2 concerning *Test Support Environment*), which is supported by our *MST-wi* toolset, and when the system under test provides a feature to configure the system time (TF8 concerning *Controllability*), which is usually feasible through secure shell connection, a feature leveraged by our toolset). For instance, an MR in SMRL can modify the HTTP-request (e.g., session IDs and cookie values) to reuse old session credentials.

Table 2.19 presents the distribution of the testability features across 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. An example is weakness *Insufficient Session Expiration*, which is associated with testability features *TF2* and *TF8* (see Table 2.7). TF2 supports the test engineer to reuse an old session (e.g., credentials or ID) by modifying the corresponding request parameters; TF8 helps to change the system time in order to invalidate this session (i.e., make it expired).

In total, we identify 102 testability features for 101 weaknesses concerning the 12 security design principles in Table 2.19. 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 to the largest subset of weaknesses (see Table 2.8). In Table 2.19, rows *Total* and *% of weaknesses* show that the two testability features with the largest number of associated weaknesses are TF1 and TF2 (22% and 40%, respectively). These two features are respectively related to testability factors *Controllability* and *test support environment*.

In our analysis, we observe that *controllability*, *test support environment* and *observability* are required to target 48%, 48% and 4% 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) [88]. We believe that this difference is due to the fact that *MST-wi* automatically simulates actions performed by a user on a Web system 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. On the other hand, the literature on testability mostly concerns functional and robustness testing, which requires a high degree of observability. The high relevance of the testability factor *Test Support Environment* for *MST-wi* is due to the fact that, to mimic a malicious user, it is

necessary to automate all the actions typically performed manually by malicious users.

Tables 2.20 and 2.21 present the testability features that enable testing for the weaknesses in the OWASP Top 10 and CWE Top 25 views, respectively. In Table 2.20, numbers are in line with the ones in Table 2.19. Indeed, the testability features that are required for testing a higher subset of weaknesses are also TF1 and TF2 in Tables 2.20. In Table 2.21, TF2 and TF3 have a significant role in testing the features.

Tables 2.19, 2.20, and 2.21 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 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 (TF1), that the testing framework supports both modifying parameter values (TF2), and that it is possible to log-in with a predefined list of credentials (TF3). Moreover, TF1, TF2, and TF3 enable test automation for highly critical weaknesses, which concern code injection, authorization, and authentication. In addition, by looking at the testability features targeting more than 10% of the vulnerabilities in Tables 2.19, 2.20, and 2.21 (i.e., at least four weaknesses associated with the OWASP Top 10 risks, two weakness in the CWE Top 25 list, and ten weaknesses concerning the security design principles), it is possible to identify a minimal set of testability features (i.e., TF1, TF2, TF3, and TF4) that should be prioritized to automatically verify both security design principles and top security risks. Since TF2 is provided by our *MST-wi* implementation and available in manual Web testing frameworks, and, further, TF1, TF3, and TF4 are common in Web systems, we conclude that MT is likely applicable in most software projects without the need for adapting existing design or testing frameworks.

To summarize, controllability and test support environment are the most required testability factors for *MST-wi*. Our results show that the most required testability features (i.e., TF1, TF2, TF3, and TF4) are either provided by *MST-wi* or available in most Web systems, thus suggesting that *MST-wi* can be applied to a wide variety of Web systems.

2.10 Empirical Evaluation

In this section, we investigate, based on two open-source case studies, the following RQs:

- ***RQ4. Is MST-wi effective?*** The goal of this research question is to assess whether *MST-wi* enables, in a reliable manner, the automated detection of security vulnerabilities.
- ***RQ5. Is MST-wi efficient?*** The goal of this research question is to analyze whether the execution time entailed by *MST-wi* is acceptable in practice.

To perform our empirical evaluation we relied on our implementation of *MST-wi*, a toolset that extends the Eclipse IDE [1]. Additional information about the toolset, including executable files, download instructions, and a screencast, is available on the tool's

website at `https://sntsvv.github.io/SMRL/`. A replicability package is provided as well [57].

2.10.1 Subjects of the Evaluation

We applied our approach, *MST-wi*, to Jenkins [77], a leading open source automation server, and Joomla [2], a popular open source content management system. The two case study subjects differ in programming languages and underlying frameworks. Jenkins is a Java Web application that we can execute within any servlet container [78]; Joomla is a PHP application that relies on the MySQL RDBMS [180] and the Apache HTTP server [32]. We chose them because they represent modern Web systems having a plug-in architecture and Web interfaces with advanced features such as Javascript-based login and AJAX interfaces. Their differences in programming languages and types of inputs may lead to different vulnerabilities and contribute to the generalizability of our empirical results. We used Jenkins version 2.121.1 and Joomla version 3.8.7. 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 [70] after June 1st, 2018. Jenkins 2.121.1 and Joomla 3.8.7 are affected by 20 and 16 vulnerabilities, respectively. Our catalog of MRs targets 40% (8 out of 20) and 31% (5 out of 16) of the vulnerabilities affecting Jenkins and Joomla, respectively. This outcome is consistent with our analysis in RQ1. Among these vulnerabilities, we selected only the ones (eight vulnerabilities for Jenkins and two for Joomla) whose effects could be manually reproduced in our environment (i.e., we could observe a security failure). For instance, we could replicate only two vulnerabilities for Joomla due to the lack of a detailed description of the attack scenarios. In addition, we considered, for Jenkins, three new vulnerabilities we discovered with *MST-wi* (one is a software fault that received a CVE identifier [127], the other two are related to Jenkins' default configuration). We have, for Joomla, one new additional configuration-related vulnerability discovered by *MST-wi*. To summarize, we considered 11 vulnerabilities for Jenkins and 3 for Joomla (see Table 3.2). In the table, we provide two CWE IDs when the CVE report refers to a generic vulnerability type (e.g., CWE_863 for incorrect authorization in reference [162]), but a more specific one fits better (e.g., CWE_280 about improper handling of privileges that may lead to incorrect authorization). We configured, for each subject, our data collection framework with multiple users having different roles. We used four credentials for Jenkins and six credentials for Joomla. We executed, for each role, the data collection framework to crawl the system under test for a maximum of 300 minutes. The data collection took 1000 minutes for Jenkins and 2280 minutes for Joomla. Crawljax completed the crawling in less than 300 minutes for the anonymous role in Jenkins and Joomla, visiting all states before the time budget was exhausted. The data collection time for Joomla was long because it has two different user interfaces (i.e., user and administrative interfaces). The crawling led to 156 and 147 input sequences for Jenkins and Joomla, respectively. Also, we implemented Selenium-based test scripts (four for Jenkins and one for Joomla) to exercise use cases not covered by Crawljax. These scripts entail a small overhead but address limitations in the crawler (cost-benefit trade-off). Based on the URLs in the input sequences collected by

Table 2.22: Vulnerabilities considered in our empirical evaluation.

Subject	Ref.	Vuln. Type	Description
Jenkins	[162]	CWE_863, CWE_280	Jenkins does not perform a permission check for URLs handling cancellation of queued builds, allowing users with Overall/Read permission to cancel queued builds.
	[163]	CWE_863, CWE_285	Jenkins does not perform a permission check for the URL that initiates agent launches, allowing users with Overall/Read permission to initiate agent launches.
	[164]	CWE_200, CWE_668	Files indicating when a plugin file was last extracted into the Jenkins `plugins/` directory are accessible via HTTP by users having Overall/Read permissions. This allows unauthorized users to determine the likely install date of a given plugin.
	[158]	CWE_22	In the file name parameter of a Job configuration, users with Job/Configure permissions can specify a relative path escaping the base directory. Such path can be used to upload a file on the Jenkins host, resulting in an arbitrary file write vulnerability.
	[166]	CWE_200	Users with Overall/Read permission are able to access the URL serving agent logs on the UI due to a lack of permission checks.
	[159]	CWE_384	Jenkins does not invalidate the existing session when a user signs up for a new user account. This allows session fixation.
	[170]	CWE_521	Jenkins does not require users to have strong passwords, which makes it easier for attackers to compromise user accounts.
	[169]	CWE_262	Jenkins does not integrate any mechanism for managing password aging; consequently, users aren't incentivized to update passwords periodically.
	[168]	CWE_79	Jenkins does not set Content-Security-Policy headers for files uploaded as file parameters to a build, resulting in a stored XSS vulnerability.
	[167]	CWE_863	Users with Overall/Read permission can access the URL used to cancel scheduled restart jobs initiated through the update center due to a lack of permission checks.
	[165]	CWE_287	Users with a valid cookie can remain logged in even if the Remember me feature has been disabled in the Jenkins configuration.
Joomla	[161]	CWE_863	Inadequate checks on the tags search fields can lead to an access level violation.
	[160]	CWE_200	Inadequate checks allow users to see the names of tags that were either unpublished or published with restricted view permission .
	[169]	CWE_262	Joomla does not integrate any mechanism for managing password aging; consequently, users aren't incentivized to update passwords periodically.

Crawljax, we determined which features in the Jenkins and Joomla documentation had been tested. We then identified the features not being tested (i.e., configuring site settings in Joomla, creating new jobs, and canceling jobs in Jenkins).

2.10.2 RQ4: Effectiveness

Experiment design

An automated testing approach is effective if it helps detect a large proportion of the faults affecting the software under test and generates a limited number of false alarms. Ideally, *MST-wi* should identify all the vulnerabilities affecting our case study subjects that can be reproduced in our environment. We executed *MST-wi* with the two case study subjects, considering all the MRs in our catalog. For each MT failure reported by *MST-wi*, we manually verified if the test input actually triggered any vulnerability (true positive).

Table 2.23: Summary of RQ4 results grouped by data collection method.

Case study	Discovered Vulnerabilities	Crawljax		Crawljax & Manual	
		Specificity	Sensitivity	Specificity	Sensitivity
Jenkins	[158, 159, 162–164, 166, 168–170]	99.94%	63.64%	99.90%	81.81%
Joomla	[160, 161, 169]	99.79%	66.67%	99.71%	100.00%
Overall	12	99.87%	64.29%	99.81%	85.71%

We measured specificity and sensitivity [116]. For specificity, we consider the set of follow-up inputs F as follows:

$$F = F_{TP} + F_{FP} + F_{TN} + F_{FN}$$

with TP, FP, TN, and FN standing for true positives (the follow-up input leads to a failure because the vulnerability has been exercised), false positives (the follow-up input leads to a failure for the wrong reason), true negatives (the follow-up input does not lead to any failure because no vulnerability has been exercised), and false negatives (the follow-up input does not lead to any failure although the vulnerability has been exercised), respectively. In our experiments, we did not observe any false negative ($F_{FN} = 0$). Indeed, we investigated the URLs known to be vulnerable and verified that they always lead to failures when they are the target of HTTP requests with parameters that exercise their vulnerability. Further, we aim to detect known vulnerabilities in the selected software versions. Therefore, we refine our definition of F as follows:

$$\begin{aligned} F &= F_{TP} + F_{FP} + F_{TN} \\ F_{TN} &= F - F_{TP} - F_{FP} \\ F_{TN} + F_{FP} &= F - F_{TP} \end{aligned}$$

Based on the above, specificity can be computed as follows:

$$\begin{aligned} \mathit{Specificity} &= \frac{F_{TN}}{F_{TN} + F_{FP}} \\ &= \frac{F - F_{TP} - F_{FP}}{F - F_{TP}} \end{aligned}$$

In other words, specificity (true negative rate) is the ratio of follow-up inputs not triggering any vulnerability that (correctly) do not lead to any MT failure.

Further, the standard definition of false positive rate leads to:

$$\begin{aligned} \mathit{FPR} &= \frac{F_{FP}}{F_{TN} + F_{FP}} \\ &= 1 - \mathit{Specificity} \end{aligned}$$

$(1 - \mathit{Specificity})$ measures the proportion of unwarranted MT failures, which enables us to discuss the effort wasted by engineers with *MST-wi*. We report both the overall specificity of *MST-wi* (across all the follow-up inputs generated in our experiment) and the specificity distribution across MRs.

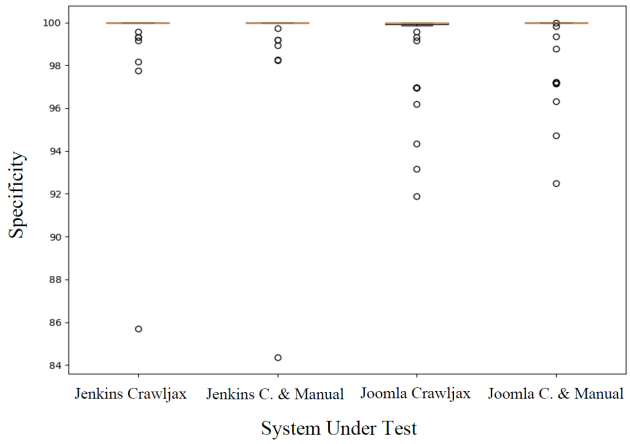

Figure 2.21: RQ4: Specificity distribution (values in Table 2.24)

Table 2.24: RQ4: Specificity distribution

	Jenkins		Joomla	
	Crawljax	C. & Manual	Crawljax	C. & Manual
First Quartile	100	100	99.94	100
Second Quartile	100	100	100	100
Third Quartile	100	100	100	100
Lower whisker	100	100	99.84	100
Upper whisker	100	100	100	100

Sensitivity (true positive rate) can be computed as

$$Sensitivity = \frac{F_{TP}}{F_{TP} + F_{FN}}$$

However, for reasons provided below, we measure *sensitivity* as the ratio of vulnerabilities discovered:

$$Sensitivity = \frac{V_{TP}}{V_{TP} + V_{FN}}$$

with V_{TP} being the number of vulnerabilities discovered and V_{FN} being the number of vulnerabilities not discovered. We employ a coarse granularity (vulnerabilities instead of follow-up inputs) because, by relying on vulnerabilities, sensitivity matches the fault detection rate, which is a standard metric to assess software testing approaches since engineers would like to know if a vulnerability is discovered, not how many times *MST-wi* reports it.

Results

Table 2.23 summarizes the results obtained with the two data collection methods supported by *MST-wi*, i.e., based on Crawljax only or integrating Crawljax and manual scripts.

We observe that the approach has **extremely high specificity** when relying on the crawler (99.87%) and when combining the crawler and manual inputs (99.81%). The high specificity indicates that only a negligible fraction of follow-up inputs leads to false alarms (121 out of 93359, 0.13%, and 103 out of 55174, 0.19%). False alarms are due to limitations in Crawljax, which, for Jenkins, did not traverse all the URLs provided by the GUI for all 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 users.

Fig. 2.21 shows boxplots presenting the distribution of specificity across MRs. Specificity is high for every MR, with the median being 100%. The lowest whisker[3] is 99.94% for Joomla with Crawljax only, which indicates that, without outliers, the minimum specificity is above 99%. In practice, for all our MRs, only a very small proportion of follow-up inputs leads to false alarms.

The worst specificity outlier is 84.38% for `OTG_INPVAL_004` with Jenkins (five false positives out of 32 follow-up inputs tested). The false positives are mainly due to asynchronous actions in the source inputs, which remain to be completed when the follow-up input is executed and thus lead to different outputs for the source and follow-up inputs, making the MR fail. However, the five false positives lead to a limited waste of developers' time. Indeed, to discover these false positives, it is sufficient to manually test the URL of the original and the follow-up action, which does not take more than ten minutes in total.

Sensitivity is high when data collection relies on both Crawljax and manual test scripts: 81.81% for Jenkins and 100% for Joomla. Since sensitivity reflects the fault detection rate (i.e., the proportion of vulnerabilities discovered), we conclude that our approach is **highly effective**. Overall, *MST-wi* detects 85.71% of the vulnerabilities targeted in our evaluation. It misses two of the eight targeted vulnerabilities in Jenkins. We can reveal one missing vulnerability only if the server configuration is modified during test execution [165]. Unfortunately, our toolset does not support the server configuration during test execution. We cannot reproduce the other missing vulnerability since it concerns the termination of Jenkins' reboot [167], which is not interruptible when Jenkins is not overloaded (our case).

When the data collection relies on Crawljax only, sensitivity drops below 70% for both Jenkins and Joomla. The low sensitivity occurs because of the incapability of our crawler to exercise some particular interactions. For example, Jenkins requires quick system interactions to exercise some features (e.g., 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). Joomla, instead, requires interactions with a widget showing all the available tags (in the presence of multiple widgets, Crawlajx may fail to systematically exercise all the widgets). However, even when the data collection is based on Crawljax only, with 9 out of 14 (64.29%) vulnerabilities detected, we nevertheless consider the overall fault detection rate satisfactory. Indeed, automatically detecting 64% of the vulnerabilities not targeted by SOTA approaches, without the need for any manual test script, is beneficial.

The benefits of *MST-wi* mostly stem 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 5 test scripts which

[3]Computed as *first quartile* $- 1.5 *$ *Inter Quartile Range*

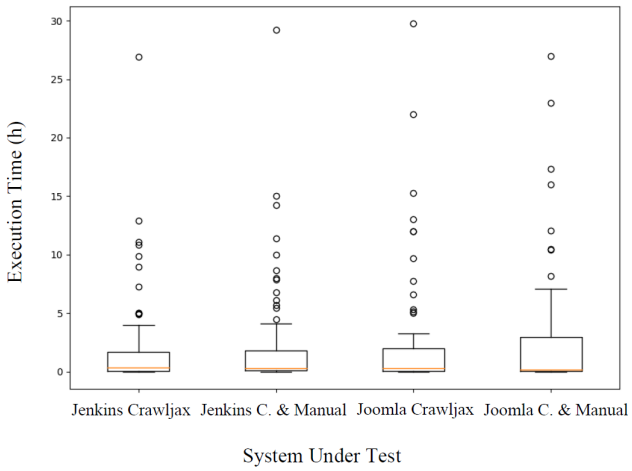

Figure 2.22: RQ5. Box-plot reporting execution time in hours (values in Table 2.25). Some outliers have been removed to ease visualization.

Table 2.25: RQ5: Distribution of execution time (hours)

	Jenkins		Joomla	
	Crawljax	C. & Manual	Crawljax	C. & Manual
First Quartile	0.06	0.11	0.05	0.084
Second Quartile	0.38	0.34	0.28	0.22
Third Quartile	2.63	2.38	2.01	3.03
Lower whisker	0.017	0.017	0.017	0.017
Upper whisker	6.48	5.80	4.95	7.44
# of MRs taking more than 14 hours	6	5	5	3

only contain 31 actions in total. This manual effort is negligible compared to 93359 input sequences (544806 actions) automatically generated by our approach to test the two systems when Crawljax and manual inputs are combined. A traditional way to verify the same scenarios would require 93359 manually implemented test scripts, each providing a distinct input sequence and a dedicated oracle (e.g., an assertion statement). Therefore, we conclude that *MST-wi* provides an advantageous cost-effectiveness trade-off compared to current practice.

2.10.3 RQ5: Efficiency

Experiment design

The execution time of *MST-wi* depends on the time required to run the crawler and the time required to execute the MRs.

The execution time of crawling depends on the number of user roles to be tested and the number of actions (i.e., inputs that can be provided through links and Web forms on different Web pages) for the SUT. However, in our context, the number of user roles has a limited impact because crawling can be parallelized. Since the development of an efficient

crawler is out of the scope of this study, we did not perform an empirical evaluation of the efficiency of our crawler (i.e., the extended Crawljax). The interested reader is referred to the original Crawljax publication for further discussions [144]. In this RQ, we discuss the time required to execute the MRs in our catalog.

We aimed to determine if *MST-wi* is efficient enough to be used in practice and what are the main factors that influence MRs' execution time. To do so, we kept track of the time required to execute each MR considered for RQ4 and discussed the execution time distribution across MRs.

By design, the time required to execute an MR is driven by the number of source inputs and follow-up inputs executed, the number of actions belonging to source and follow-up inputs, and the complexity of the MR executed (e.g., the number of instructions in the MR and computational complexity of the functions invoked by the MR). Finally, the hardware components used to run the Web server can have different response times and, consequently, affect the execution time of the MR.

Based on the above, we studied the correlation between execution time and the number of source inputs, follow-up inputs, and actions in follow-up inputs by relying on the non-parametric Spearman's correlation coefficient. As for hardware, we did not study the effect of different hardware configurations on *MST-wi*efficiency but executed our experiments in scenarios that we consider feasible for testing software with *MST-wi*: a virtual machine installed on professional desktop PCs (Dell G7 7500, RAM 16Gb, Intel(R) Core(TM) i9-10885H CPU @ 2.40GHz) and terminal access to a shared remote server with Intel(R) Xeon(R) Gold 6234 CPU (3.30GHz) and 8 CPU cores.

Results

Fig. 2.22 provides boxplots reporting the execution time distribution, in hours, for the MRs executed to address RQ4. The highest third quartile is 3.03 hours and indicates that 75% of the MRs can be executed for both systems overnight or in half a working day (i.e., less than four hours) if we parallelize the executions of the MRs (e.g., by running each on a dedicated virtual environment). Further, when excluding outliers, the maximum execution time is about seven hours and a half for Joomla tested with manual and Crawljax-based inputs (see *Upper whisker* in Table 2.25), which indicates that most MRs can be executed overnight. Considering that security testing is currently performed manually and is error-prone, we believe that the cost of setting up parallelized execution environments is justified.

From Table 2.25, we can see that, across configurations (i.e., executing *MST-wi* with inputs derived by relying on Crawljax only or by combining Crawljax and manual inputs), between four and six MRs could not be executed overnight because they took more than 14 hours. These MRs look for injection, authorization, and authentication problems by relying on a catalog of crafted values passed to URL parameters or form inputs. In practice, they test the same URL multiple times with different users and inputs, leading to a combinatorial explosion. However, in our framework, it is possible to parallelize the execution of a single MR across multiple nodes (e.g., virtual machines running on a Cloud or grid infrastructure); each node iterates over a subset of source inputs. With ten execution

Table 2.26: RQ5: Spearman correlation

Set of features\System under test	Jenkins				Joomla			
	Crawljax		Crawljax & Manual		Crawljax		Crawljax & Manual	
	Spearman	P-value	Spearman	P-value	Spearman	P-value	Spearman	P-value
Execution time and number of source inputs	0.39	0.00076	0.38	0.00071	0.16	0.20	0.17	0.18
Execution time and number of follow-up inputs	0.52	1.91e-06	0.36	0.0017	0.56	4.15e-07	0.55	7.15e-07
Execution time and number of actions in follow-up inputs	0.51	4.19e-06	0.43	0.00013	0.52	3.41e-06	0.52	5.72e-06

nodes for each of these six MRs, it should be feasible to execute them overnight. For example, we parallelized the execution of the MR leading to the worst-case execution time (i.e., `CWE_138..._OTG_AUTHZ_001b` in Jenkins) and observed a maximum execution time of 14 hours, thus confirming our assumption.

Table 2.26 reports, for our experiments, the Spearman's coefficients capturing the correlation between execution time and the number of (a) source inputs, (b) follow-up inputs, and (c) actions in follow-up inputs. We observe that, as expected, the execution time significantly correlates (i.e., coefficient above 0.5 and p-value below 0.05) with the number of actions in follow-up inputs; indeed, each action leads to a request execution in the browser. We also note a correlation with the number of follow-up inputs, which is lower in the case of Jenkins. Such low correlation for Jenkins is observed when testing with source inputs derived from both Crawljax and manual test cases. Indeed, manual test cases include a smaller set of actions than the ones collected by the crawler, but they test URLs that are also covered by the source inputs derived by the crawler. Consequently, since several MRs do not test the same URL twice, *MST-wi*, when data collection is based on Crawljax and manual test cases, tends to generate follow-up inputs that are shorter and, therefore, quicker to execute (the long follow-up inputs derived from the Crawljax source inputs are not executed because they include URLs already tested by relying on the short manual source inputs). We do not observe the same trend in Joomla, likely because the crawled source inputs contain fewer actions (the average number of actions for each source input is 5.9 for Joomla and 6.7 for Jenkins). Finally, the correlation between the execution time and the number of source inputs is more limited for Jenkins and not significant for Joomla. This limited correlation is mainly due to source inputs not leading to follow-up inputs (e.g., because a precondition does not hold) and, consequently, causing execution times that vary a lot across MRs even if the number of source inputs processed by these MRs is the same. Concluding, efficiency does not depend on the number of input interfaces but on the complexity of the features of the Web-system under test. The former drives the number of source inputs generated (low correlation with execution time), and the latter drives the length of the source and follow-up inputs (higher correlation with execution time). We selected well-known case study subjects representative of a broad set of systems; engineers testing systems not similar to Jenkins and Joomla may observe different efficiency results.

2.11 Threats to Validity

Internal validity

Two co-authors inspected all the results to identify applicable SOTA oracle automation

strategies for security testing (RQ1), to determine what MR can target a vulnerability type (RQ2), and to analyze the causes undermining the applicability (RQ2) and testability factors enabling *MST-wi* (RQ3). For RQ1, one researcher proposed oracle types always confirmed by the second researcher. For RQ2, one researcher provided an MR or indicated the reason for not being able to apply *MST-wi*. The second researcher verified the implemented MR and proposed alternative implementations (when necessary). Regarding the reasons for not applying *MST-wi*, the second researcher provided alternatives when in disagreement. By considering the category assigned by each researcher to each weakness considered for RQ2, we computed the Cohen's Kappa coefficient to measure inter-rater agreement, leading to a Kappa equal to 0.707, with a 95% confidence interval between 0.611 and 0.804, which indicates *substantial agreement*. Disagreement is impossible in RQ3 since the identification of testability factors is based on the utility functions used by the MR. Once the first researcher assigned testability factors to each MR, the second researcher verified the correctness of the outcome.

To minimize *implementation errors* in RQ4 and RQ5, we carefully inspected and tested the *MST-wi* toolset before running our experiments. Also, we executed our MRs on DVWA [240], a security benchmark, thus ensuring that our MRs can discover some of the targeted vulnerabilities (injection vulnerabilities of different kinds, in this case).

Conclusion validity

For RQ1, RQ2, and RQ3, we reported the proportion of items in a given catalog that belong to a certain class: types of oracles (RQ1), feasible MRs (RQ2), inapplicability causes and testability features (RQ3). For RQ4, we discussed specificity and sensitivity. Therefore, statistical tests were not required for RQ1, RQ2, RQ3, and RQ4.

For RQ5, the underlying distribution of the data (i.e., execution time and the number of executed source inputs, follow-up inputs, and actions in follow-up inputs) is not known in our context. Therefore, we relied on Spearman's rank correlation, which is non-parametric, to avoid violating the assumptions of parametric tests for correlation analysis.

For RQ4 and RQ5, the sources of *randomness affecting results* might be (i) the workload of the machines used to run the experiments (slowing down the performance of both *MST-wi* and the case study subjects) and (ii) the presence of other users interacting with the software under test (affecting both execution time and system outputs). To mitigate these effects, we run the experiments in dedicated environments with the study subjects used only by our framework.

Construct validity

We discuss *construct validity* in terms of face, content, convergent, and predictive validity [198].

For RQ4 and RQ5, the constructs we considered in our work are effectiveness and efficiency. Effectiveness is measured through two reflective indicators, which are sensitivity

and specificity. Efficiency is measured in terms of execution time. Concerning *face validity*, we believe our indicators are appropriate. Indeed, sensitivity is the ratio of vulnerabilities discovered and corresponds to the fault detection rate, commonly used in software testing papers to evaluate testing effectiveness. Specificity captures the proportion of follow-up inputs not leading to failures (i.e., a large majority for stable systems like Jenkins and Joomla) that, correctly, do not trigger any failure report in *MST-wi*. Specificity captures developers' time wasted on investigating false alarms. Indeed, this is proportional to the number of follow-up inputs checked in vain. Execution time is a direct measure that enables us to assess if, for systems similar to our case study subjects, *MST-wi* is efficient enough to be integrated into software development.

Content validity concerns the breadth of the construct. High sensitivity (i.e., fault detection rate) is a condition for a software testing technique to be useful. However, it does not imply that engineers can understand the root cause of a failure. Successful root cause analysis depends on other factors, such as the software engineer's experience and knowledge about the software under test and the availability of appropriate debugging and logging tools, which are out of the scope of this study. The false positive rate, a measure of the effect of false alarms, is equal to *1 - specificity* and is complementary to sensitivity; therefore, we believe our choice of measurements to be complete. To discuss efficiency, we rely on the number of source inputs, follow-up inputs, and actions in follow-up inputs; they capture the complexity of the SUT since they reflect the actions that might be performed by an attacker and are automatically derived through a crawler exercising the system under test. Other metrics for the size of the SUT (e.g., lines of code, number of Web pages) may not capture its complexity (e.g., knowing the number of Web pages is insufficient to understand how many form inputs might be submitted to the SUT). Therefore, we believe that studying the correlation between our selected complexity metrics and execution time is adequate to discuss the efficiency of the approach.

As for *predictive validity*, we reported statistics for RQ5 based on a non-parametric correlation coefficient (Spearman's) because the collected data (i.e., number of follow-up inputs, number of actions in follow-up inputs, and execution time) does not appear to be normally distributed.

External validity

In RQ1 and RQ2, we considered specific catalogs (the *OWASP testing book*, the *CWE view for common security architectural tactics*, the CWE view for *OWASP Top 10*, and the CWE view for *CWE Top 25*) that include a set of vulnerability types considered broad and complete by security researchers [81]. In RQ3, we derived our results from the characteristics of the implemented MRs (i.e., what enables their execution). Since our MRs are not system-specific, our testability features apply to any Web system.

To strengthen the generalizability of our conclusions in RQ4, we selected systems that are representative of modern Web systems but different in terms of technical and process aspects. For RQ5, we executed our experiments by relying on hardware commonly available to Web and security engineers (i.e., laptops, virtual machines, and web servers). Contrary

to our initial *MST-wi* study [132], we did not analyze the Web system developed in the context of the EDLAH2 [79] project because the project ended and our license to access the system expired. However, the interfaces exposed by Joomla are more complex than the ones of the EDLAH2 system (e.g., a larger set of accessible Web pages).

2.12 Related Work

This section covers the related work across three categories: (i) *software security testing*, (ii) *model-based security testing*, and (iii) *metamorphic testing*. In the first category, we present existing security testing strategies and discuss how *MST-wi* complements them. Model-based security testing is one of the standard security testing methods [85, 229]. Although it is a relatively new research field, many model-based approaches have been published lately, and we, therefore, present and compare them with *MST-wi*. Finally, in the last category, we discuss MT techniques since they represent the foundation on which *MST-wi* has been defined.

2.12.1 Software Security Testing

Security testing approaches can be categorized [84] as follows: (i) security functional testing validating whether the specified security properties are implemented correctly and (ii) security vulnerability testing mimicking attacks that target typical system vulnerabilities. *MST-wi* can be applied to both security functional and vulnerability testing since MRs can capture security properties (e.g., a login screen should always be shown after a session timeout) and the properties of the inputs and outputs involved in discovering a vulnerability (e.g., the output generated when requesting an admin resource without authentication should differ from the one obtained with authentication).

Many security vulnerability testing approaches rely on an implicit test oracle, i.e., one that relies on tacit knowledge to distinguish between correct and incorrect system behavior [91]. It is the case for approaches targeting buffer overflows, memory leaks, unhandled exceptions, and denial of service [48, 179, 225], which mostly rely on mutational fuzzing [250], i.e., the generation of new inputs through the random modification of existing ones. For instance, Bekrar et al. [48] present a technique combining mutational fuzzing with data tainting and coverage analysis to identify security vulnerabilities in file processors and network protocols. Implicit oracles deal with simple abnormal system behavior, such as unexpected system termination, and are system-agnostic. The literature, however, lacks explicit oracles for vulnerabilities leading to invalid outputs (e.g., providing data to a user who is not supposed to access it); indeed, such outputs are system-specific and difficult to capture with the mechanisms used for implicit oracles (e.g., runtime exceptions).

Vulnerability testing approaches for code injections also suffer from the oracle problem [33, 34, 46, 106, 138, 197, 202, 230]. For instance, test cases targeting SQL injections are in the form of HTTP requests that trigger responses from a Web application while a crawler receives responses in which some predefined keywords (e.g., "invalid") are searched [197].

When no keywords are detected, the crawler cannot determine whether the injection is filtered by the system under test. To resolve this problem, Huang et al. [96] proposed an MT-like technique that 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 target a broader set of security vulnerabilities are missing. In contrast, *MST-wi* targets a wide variety of security vulnerabilities (including code injection vulnerabilities); a detailed analysis of security vulnerabilities that can and cannot be addressed by our approach was provided in Section 2.9.

2.12.2 Model-based Security Testing

Model-based approaches [83, 85] mostly target security vulnerability testing (e.g., References [50, 53, 94, 101–104, 134, 137, 237, 238, 243, 245, 246]), whereas some solutions target security functional testing (e.g., References [119,135,136,139,171,172,244]). Most of them only generate test sequences from security models and do not address the oracle problem. For instance, Marback et al. [134] propose a model-based security testing approach that automatically generates security test sequences from threat trees.

Model-based approaches that generate test cases with oracles [244–246] 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). For instance, Xu et al. [245, 246] automatically generate executable vulnerability test cases from formal threat models. Further, model-level test oracles (tokens in markings of attack paths) are directly mapped to implementation-level code. But such mapping is not feasible when it is difficult to specify precise test oracles and automatically compare expected values to the actual results. The same problem also affects another approach that targets access control policies [244]. Overall, these approaches do not free engineers from significant implementation effort since they require the manual implementation of the executable oracle code.

2.12.3 Metamorphic Testing

With MT, we aim to address the limitations of security testing approaches described above. Indeed, MT supports oracle automation thanks to MRs that can precisely capture the relations between test inputs and outputs. In practice, MT frees engineers from implementing a specific oracle (e.g., test case assertions) for each test case. Considerable research has been devoted to developing MT approaches for various domains such as computer graphics (e.g., [89,105,115,140]), simulation (e.g., [67,75,176]), Web services (e.g., [59,222,257]), embedded systems (e.g., [58,100,114,231]), compilers (e.g., [118,227]), variability and decision support (e.g., [113, 206, 211, 212]), bioinformatics (e.g., [65, 196, 199]), numerical programs (e.g., [38, 64]), and machine learning (e.g., [174, 242]). However, very little attention has been paid to its application in security testing [210].

Preliminary applications of MT to security testing focus on the functional testing of security components (i.e., testing encryption programs in the absence of oracles [223], verifying the output of code obfuscators and the rendering of login interfaces [62]), and the verification of low-level properties broken by specific security bugs (e.g., the heartbleed bug [224] which affects the relation between the size of the payload data field of an SSL message and the length declared in the same message). Although these works demonstrate the feasibility of MT for security testing, 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. In contrast, *MST-wi* supports the specification of MRs for many vulnerabilities and automates the generation of executable metamorphic test cases from the MRs.

Although MT is highly automatable, MT research has mostly focused on the application of MT to specific testing problems [210]. For instance, Kuo et al. [114] report on a case study using metamorphic testing to detect faults in a wireless metering system. Ding et al. [75] present a case study for fault detection in a Monte Carlo modeling program simulating photon propagation. Chen et al. [64] focus on applying MT to programs implementing partial differential equations. These works do not provide any systematic method to specify MRs and proper tool support. In general, few approaches provide tool support enabling engineers to write system-level MRs [210]. Those which require that MRs be defined either as Java methods [258] or method pre-/post-conditions [173] limit the adoption of MT to verify system-level security properties; indeed, they target single methods and not the output provided by the system as a whole. Since MRs often employ a declarative notation, engineers need significant, additional effort to translate abstract, declarative MRs into an imperative programming language. To avoid such overhead, we propose a DSL as part of *MST-wi* (see Section 2.4). Segura et al. [207,208] provide a template-based approach for describing MRs. The proposed template aims to ease communication among practitioners but does not support security-related language constructs. Further, there is no automated support to transform template-based MRs into executable test cases.

The notion of *metamorphic property* (e.g., the *permutative* property that specifies that the order of inputs should not affect the output) introduced by Murphy et al. [175] forms the basis for *general metamorphic relations*, which are analogous to metamorphic relation patterns. Zhou et al. [256] define the notion of metamorphic relation pattern (MRP) as *an abstraction that characterizes a set of (possibly infinitely many) metamorphic relations*. Subclasses of MRPs are proposed in the literature: *metamorphic relation input pattern (MRIP)* [256] and *metamorphic relation output pattern (MROP)* [213]. An MRIP is an abstraction characterizing the relations among the source and follow-up inputs of a set of metamorphic relations; an MROP describes an abstract relation among the source and follow-up outputs.

Segura et al. [213] propose six MROPs (equivalence, equality, subset, disjoint, complete, and difference) for testing RESTful web APIs (implementing create, read, update, or delete operations over a resource). In our MR catalog, we leverage some of these patterns to define output conditions; in particular, our MRs verify equality, difference, and subset (i.e., what we achieve with `userCanRtrieveContent`, which checks if the output is a subset of what was already observed in previous executions). Segura et al. [209] also define a catalog of

MRPs for query-based systems, which focus on the properties of inputs being conditions (e.g., the keywords used when executing a search engine, which can be joined) and outputs being data sets (e.g., the results returned by the search engine, which can be disjoint, subsets, or shuffled). Zhou et al. [256] propose a *symmetry* MRP and a *change direction* MRIP to test the system from "different viewpoints", e.g., checking if an object recognition system recognizes the same object regardless of whether it is played forward or backward (changing direction). The works described above assume that source inputs are either single items or item sets, which simplifies the definition of patterns capturing mathematical properties (e.g., symmetry or equality). In our work, we focus instead on source inputs and outputs that are action sequences and corresponding output sequences; action sequences are necessary to describe interactions with complex systems. Consequently, our patterns capture the different operations to be performed on these input/output sequences. The patterns provided in the literature are part of our MRs, but they capture only output conditions, as described above.

2.12.4 Summary

In Table 2.27, based on a set of features necessary for security testing, we summarize the differences between *MST-wi* and related work. For each approach, the symbol '+' indicates that the approach provides the feature, '-' indicates that it does not, and 'NA' indicates that the feature is not applicable because it can be implemented only by approaches in other categories (i.e., Metamorphic Testing, Model-based Security Testing, or other Software Security Testing approaches). For instance, Ognawala et al. [179] employ symbolic execution to detect memory out-of-bounds/buffer overflow vulnerabilities caused by unhandled memory operations. Therefore, none of the features related to MT, such as the support for specifying MRs, are considered for Ognawala et al. [179], as depicted in Table 2.27. Most automated security testing approaches do not address the oracle problem [33,34,46,106,138,197,202,230] or rely on an implicit test oracle [48,179,225]. The few approaches that do address the oracle problem focus on a limited number of security vulnerabilities [96]. Also, most model-based security testing approaches do not address the oracle problem since they only generate test sequences from security models [120, 134, 237, 238]. Some model-based approaches derive test cases with oracles [244–246] but require mappings between model-level abstractions and executable code implementing the oracle logic. MT can overcome these limitations, but existing MT solutions target a few specific security vulnerabilities and do not support automated testing based on MRs capturing general security properties. To overcome these limitations, we need a dedicated DSL and algorithms that automate the execution of MT. To the best of our knowledge, *MST-wi* is the only approach that supports, with a DSL, the specification of MRs capturing a wide range of security properties, automates the generation of executable metamorphic test cases from the MRs, and automatically detects various vulnerabilities based on those relations.

Table 2.27: Summary and comparison of related work.

		Support for various vulnerabilities	**No need for implicit oracles**	**Model-based test generation including oracle**	**No need for model-based mappings or manual oracle implementation**	**Application of MT to security testing**	**DSL support to specify MRs**	**Publicly Available Tool support for MT**
	MST-wi	+	+	*NA*	+	+	+	+
Software Security Testing	Ognawala et al. [179]	−	−	*NA*	*NA*	*NA*	*NA*	*NA*
	Bekrar et al [48]	+	−	*NA*	*NA*	*NA*	*NA*	*NA*
	Takanen et al. [225]	+	−	*NA*	*NA*	*NA*	*NA*	*NA*
	Kals et al. [106]			*NA*	*NA*	*NA*	*NA*	*NA*
	Martin et al. [138]	−	+	*NA*	*NA*	*NA*	*NA*	*NA*
	Bau et al. [46]	−	+	*NA*	*NA*	*NA*	*NA*	*NA*
	Appelt et al. [34]	−	+	*NA*	*NA*	*NA*	*NA*	*NA*
	Salas et al. [202]	−	+	*NA*	*NA*	*NA*	*NA*	*NA*
	Tripp et al. [230]	−	+	*NA*	*NA*	*NA*	*NA*	*NA*
	Appelt et al. [33]	−	+	*NA*	*NA*	*NA*	*NA*	*NA*
	Huang et al. [96]	−	+	*NA*	*NA*	*NA*	*NA*	*NA*
Model-based Security Testing	Le Traon et al. [119]	*NA*	+	−	−	*NA*	*NA*	*NA*
	Mouelhi et al. [171, 172]	*NA*	+	−	−	*NA*	*NA*	*NA*
	Martin et al. [135]	*NA*	+	−	−	*NA*	*NA*	*NA*
	Martin et al. [136]	*NA*	+	−	−	*NA*	*NA*	*NA*
	Wimmel and Jürjens [238]	*NA*	+	−	−	*NA*	*NA*	*NA*
	Masood et al. [139]	*NA*	+	−	−	*NA*	*NA*	*NA*
	Bertolino et al. [50]	+	+	−	−	*NA*	*NA*	*NA*
	Blome et al. [53]	+	−	−	+	*NA*	*NA*	*NA*
	He et al. [94]	+	+	−	−	*NA*	*NA*	*NA*
	Marback et al. [134]	+	+	−	−	*NA*	*NA*	*NA*
	Jürjens et al. [104]	+	+	−	−	*NA*	*NA*	*NA*
	Xu et al. [243]	+	+	−	−	*NA*	*NA*	*NA*
	Lebeau et al. [120]	+	+	−	−	*NA*	*NA*	*NA*
	Whittle et al. [237]	+	+	−	−	*NA*	*NA*	*NA*
	Xu et al. [245, 246]	+	+	+	−	*NA*	*NA*	*NA*
	Xu et al. [244]	*NA*	+	+	−	*NA*	*NA*	*NA*
Metamorphic Testing	Mayer and Guderlei [140]	*NA*	+	*NA*	+	−	−	−
	Just and Schweiggert [105]	*NA*	+	*NA*	+	−	−	−
	Kuo et al. [115]	*NA*	+	*NA*	+	−	−	−
	Jameel et al. [99]	*NA*	+	*NA*	+	−	−	−
	Chan et al. [60, 61]	*NA*	+	*NA*	+	−	−	−
	Chen et al. [67]	*NA*	+	*NA*	+	−	−	−
	Ding et al. [75]	*NA*	+	*NA*	+	−	−	−
	Murphy et al. [176]	*NA*	+	*NA*	+	−	−	−
	Sim et al. [215]	*NA*	+	*NA*	+	−	−	−
	Chan et al. [59]	*NA*	+	*NA*	+	−	−	−
	Sun et al. [222]	*NA*	+	*NA*	+	−	−	+
	Zhou et al. [257]	*NA*	+	*NA*	+	−	−	+
	Tse et al. [231]	*NA*	+	*NA*	+	−	−	−
	Chan et al. [58]	*NA*	+	*NA*	+	−	−	−
	Kuo et al. [114]	*NA*	+	*NA*	+	−	−	−
	Jiang et al. [100]	*NA*	+	*NA*	+	−	−	−
	Tao et al. [227]	*NA*	+	*NA*	+	−	−	+
	Yao et al. [248]	*NA*	+	*NA*	+	−	−	−
	Segura et al. [211,212]	*NA*	+	*NA*	+	−	−	+
	Kuo et al. [113]	*NA*	+	*NA*	+	−	−	−
	Chen et al. [65]	*NA*	+	*NA*	+	−	−	−
	Pullum et al. [196]	*NA*	+	*NA*	+	−	−	−
	Chen et al. [64]	*NA*	+	*NA*	+	−	−	−
	Aruna and Prasad [38]	*NA*	+	*NA*	+	−	−	−
	Xie et al. [242]	*NA*	+	*NA*	+	−	−	−
	Murphy et al. [174]	*NA*	+	*NA*	+	−	−	−
	Segura et al. [213]	*NA*	+	*NA*	+	−	−	+
	Segura et al. [207,208]	*NA*	+	*NA*	+	−	+	−
	Chen et al. [62]	−	+	*NA*	+	+	−	−
	Sun et al. [223]	−	+	*NA*	+	+	−	−
	Luu et al. [126]	*NA*	+	*NA*	+	−	−	−
	Lascu et al. [117]	*NA*	+	*NA*	+	−	−	+
	Ayerdi et al. [42]	*NA*	+	*NA*	+	−	−	+
	Xu et al. [247]	*NA*	+	*NA*	+	−	−	−

2.13 Conclusion

In this study, we presented an approach, *MST-wi*, that enables engineers to specify metamorphic relations 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 catalog of MRs inspired by OWASP guidelines and vulnerability descriptions in the CWE [26], (3) a data collection framework crawling the system under test to derive input data automatically, and (4) a testing framework automatically performing security testing based on the MRs and the input data [55].

Our analysis of the OWASP guidelines shows that our approach can automate 39% of the security testing activities not currently targeted by SOTA techniques, indicating that it significantly contributes to addressing the oracle problem in security testing. Further, our catalog of MRs can detect 101 vulnerability types in the CWE view for security design principles (45% of the total), thus highlighting the broad applicability of *MST-wi* in the security testing context.

Our empirical results with two open-source Web systems show that the approach requires limited manual effort and detects 85.71% of the targeted vulnerabilities, thus suggesting it is highly effective. Moreover, since at most 0.19% of the executed follow-up inputs led to a false alarm, the impact of false alarms is minimal. Finally, the execution of our MRs can be parallelized, thus enabling metamorphic security testing to be automatically performed overnight.

Chapter 3

AIM: Automated Input Set Minimization

This chapter addresses TO_2, which aims to reduce the cost of metamorphic security testing by minimizing the set of source inputs processed by metamorphic relations, while preserving the ability of MRs to detect security vulnerabilities. The contents of this chapter have been published in the Journal of *Transactions on Software Engineering (TSE)*.

3.1 Overview

As discussed in Chapter 1 and Chapter 2, Web systems, from social media platforms to e-commerce and banking systems, are a backbone of our society: they manage data that is at the heart of our social and business activities (e.g., public pictures, bank transactions), and, as such, should be protected. To verify that Web systems are secure, engineers perform security testing, which consists of verifying that the software adheres to its security properties (e.g., confidentiality, availability, integrity, and authenticity). Such testing is typically performed by simulating malicious users interacting with the system under test [129, 130]. In security testing, when a software output differs from the expected one, then a software vulnerability (i.e., a fault affecting the security properties of the software) has been discovered.

Automatically deriving test oracles for the software under test (SUT) is called the *oracle problem* [43], which entails distinguishing correct from incorrect outputs for all potential inputs. Except for the verification of basic reliability properties—ensuring that the software provides a timely output and does not crash—the problem is not tractable without additional executable specifications (e.g., method post-conditions or detailed system models), which, unfortunately, are often unavailable. Further, since software vulnerabilities tend to be subtle, it is necessary to exercise each software interface with a large number of inputs (e.g., providing all the possible code injection strings to a Web form). When a large number of test inputs are needed, even in the presence of automated means to generate them (e.g., catalogs of code injection strings), testing becomes impractical if we lack solutions to automatically derive test oracles.

Metamorphic testing was proposed to alleviate the oracle problem [210] by testing not the input-output behavior of the system, but by comparing the outputs of multiple test executions [47,210]. It relies on metamorphic relations, which are specifications expressing how to derive a follow-up input from a source input and relations between the corresponding outputs. Such an approach has shown to be useful for security testing, also referred to as metamorphic security testing (MST) [47,132]. MST consists in relying on MRs to modify source inputs to obtain follow-up inputs that mimic attacks and verify that known output properties captured by these MRs hold (e.g., if the follow-up input differs from the source input in some way, then the output shall be different). For instance, one may verify if URLs can be accessed by users who should not reach them through their user interface, thus enabling the detection of authorization vulnerabilities.

MST has been successfully applied to testing Web interfaces [47, 132] in an approach called *MST-wi* [47]; in such context, source inputs are sequences of interactions with a Web system and can be easily derived using a Web crawler. For example, a source input may consist of two actions: performing a login and then requesting a specific URL appearing in the returned Web page. *MST-wi* integrates a catalog of 76 MRs enabling the identification of 101 vulnerability types.

The performance and scalability of metamorphic testing naturally depends on the number of source inputs to be processed. In the case of *MST-wi*, we demonstrated that scalability can be achieved through parallelism; however, such a solution may not fit all development contexts (e.g., not all companies have an infrastructure enabling parallel execution of the software under test and its test cases). Further, even when parallelization is possible, a reduction of the test execution time may provide tangible benefits, including earlier vulnerability detection. In general, what is required is an approach to minimize the number of source inputs to be used during testing.

In this work, we address the problem of minimizing source inputs used by MRs to make metamorphic testing more scalable, with a focus on Web systems, though many aspects are reusable to other domains. We propose the Automated Input set Minimizer (AIM) approach, which aims at minimizing a set of source inputs (hereafter, the *initial input set*), while preserving the capability of MRs to detect security vulnerabilities. More in detail, this work includes the following contributions:

- We propose AIM, an approach to minimize input sets for metamorphic testing while retaining inputs able to detect vulnerabilities. Note that many steps of AIM are not specific to Web systems while others would need to be tailored to other domains (e.g., desktop applications, embedded systems). This approach includes the following novel components:
  - An extension of the *MST-wi* framework to retrieve output data and extract cost information about MRs without executing them.
  - A black-box approach leveraging clustering algorithms to partition the initial input set based on security-related characteristics.

  - MOCCO (Many-Objective Coverage and Cost Optimizer), a novel genetic algorithm which is able to efficiently select diverse inputs while minimizing their total cost.
  - IMPRO (Inputset Minimization Problem Reduction Operator), an approach to reduce the search space to its minimal extent, and then divide it in smaller independent parts.

- We provide a prototype framework for AIM [56], integrating the above components and automating the process of input set minimization for Web systems.

- We report on an extensive empirical evaluation (about 800 hours of computation) aimed at assessing the effectiveness of AIM in terms of vulnerability detection and performance, considering 18 different AIM configurations and 4 standard search algorithms for security testing, on the Jenkins and Joomla systems, which are the most used Web-based frameworks for development automation and context management.

Chapter Structure. In Section 3.2, we introduce background information necessary to state our problem and detail our approach. In Section 3.3, we define the problem of minimizing the initial input set while retaining inputs capable of detecting distinct software vulnerabilities. Next, we present an overview of AIM in Section 3.4 and then detail our core technical solutions in Sections 3.5 to 3.9. We report on a large-scale empirical evaluation of AIM in Section 3.10, and address the threats to the validity of the results in Section 3.11. We discuss and contrast related work in Section 3.12, and draw conclusions in Section 3.13.

3.2 Background

In this section, we present the concepts required to define our approach. We first provide a brief background on MST (Section 3.2.1). Next, we briefly describe three clustering algorithms: K-means (Section 3.2.3), DBSCAN (Section 3.2.3), and HDBSCAN (Section 3.2.3). Finally, we introduce optimization problems (Section 3.2.4).

3.2.1 Metamorphic Security Testing

In the previous chapter, we defined the concept of MST and introduced a tool, *MST-wi*, to automate the MST process for Web applications. *MST-wi* enables software engineers to define MRs that capture the security properties of Web systems. *MST-wi* includes a data collection framework that crawls the Web system under test to automatically derive source inputs. Each source input is a sequence of interactions of the legitimate user with the Web system. Moreover, *MST-wi* includes a Domain Specific Language (DSL) to support writing MRs for security testing. Finally, *MST-wi* provides a testing framework that automatically performs security testing based on the defined MRs and the input data.

An MR tests the initial set of source inputs, with different URLs and users, and transforms each one several times with different parameters, e.g., system file paths, leading to a combinatorial explosion. The more executed actions, the longer the execution time. The provided MR, with an input set of 160 source inputs on Jenkins, executed more than 200 000 follow-up inputs in 17 694 minutes (about 12 days) on a professional desktop PC (Dell G7 7500, RAM 16Gb, Intel(R) Core(TM) i9-10885H CPU @ 2.40GHz). Even when parallelization is possible, a reduction of the test execution time may provide tangible benefits, including earlier vulnerability detection. This warrants an approach to minimize the initial set of source inputs, based on the cost of each input.

However, knowing the execution time of each source input would require to execute them on the considered MRs, hence defeating the purpose of input set minimization. To avoid executing MRs but relying on the number of actions executed by an MR as a surrogate metric for execution time to guide the minimization technique, we computed the Spearman's correlation coefficients between execution time and number of actions exercised by the MRs tested in a previous study [47]. It led to significant correlations (i.e., coefficients of correlation above 0.5 and p-value p below 0.05), thus confirming the feasibility of relying on the number of actions as a surrogate metric for execution time.

The example in Figure 3.1 depicts a typical linear correlation between the execution time of an MR and the number of executed actions. It uses randomly selected source inputs and an MR written for CWE 863. Each point represents the execution time (x-axis) and the number of executed actions (y-axis) for a given input. The linear regression is represented by the blue line and the corresponding coefficient of determination is 97.8%, indicating a strong linear correlation.

3.2.2 Test Suite Minimization

Test suites are prone to redundant test cases that, if not removed, can lead to a massive waste of time and resources [249], thus warranting systematic and automated strategies to eliminate redundant test cases, that are referred to as test suite *minimization*.

While test suite minimization techniques are very diverse [249], most of them are white-box approaches aiming at minimizing the size of the test suite while maximizing code coverage. For instance, several test minimization approaches used greedy heuristics to select test cases based on their code coverage [148, 178]. Black-box approaches include the FAST-R family of scalable approaches that leverages a representation of test source code (or command line inputs) in a vector-space model [69] and the ATM approach that is based on the abstract syntax tree of test source code [189]. Both use similarity metrics to cluster and then select test cases.

In the context of Web systems, both the system source code and test code are not available to determine similarity between source inputs, warranting a different black-box approach able to cluster and select source inputs for these systems. Moreover, to make MST scalable, MR execution time should be minimized while preserving vulnerability detection, warranting an approach that minimizes the number of executed actions (Section 3.2.1) while covering all the input clusters.

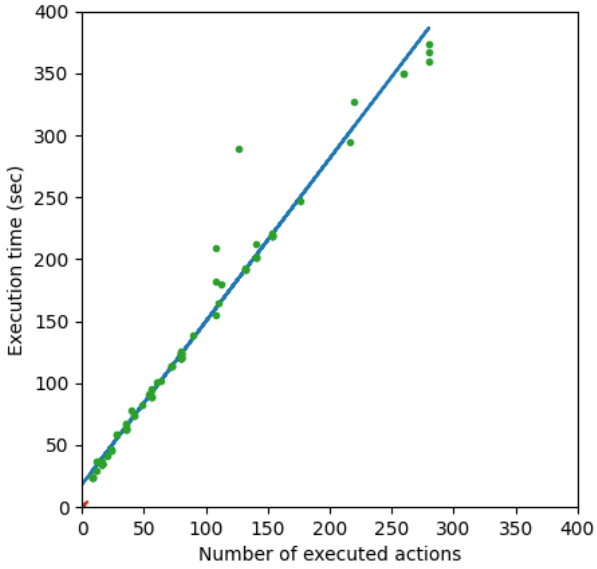

Figure 3.1: Linear regression between the number of executed actions and the execution time of a metamorphic relation. Each input is represented by a green dot, while the blue line depicts the linear regression model.

3.2.3 Clustering

Within the clustering steps in Section 3.6, we rely on three well-known clustering algorithms: K-means (Section 3.2.3), DBSCAN (Section 3.2.3), and HDBSCAN (Section 3.2.3).

K-means

K-means is a clustering algorithm which takes as input a set of data points and an integer k. K-means aims to assign data points to k clusters by maximizing the similarity between individual data points within each cluster and the center of the cluster, called centroid. The centroids are randomly initialized, then iteratively refined until a fixpoint is reached [37].

DBSCAN

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is an algorithm that defines clusters using local density estimation. This algorithm takes as input a dataset and two configuration parameters: the distance threshold ϵ and the minimum number of neighbors n.

The distance threshold ϵ is used to determine the ϵ-neighborhood of each data point, i.e., the set of data points that are at most ϵ distant from it. There are three different

types of data points in DBSCAN, based on the number of neighbors in the ϵ-neighborhood of a data point:

Core If a data point has a number of neighbors above n, it is then considered a core point.

Border If a data point has a number of neighbors below n, but has a core point in its neighborhood, it is then considered a border point.

Noise Any data point which is neither a core point nor a border point is considered noise.

A cluster consists of the set of core points and border points that can be reached through their ϵ-neighborhoods [82]. DBSCAN uses a single global ϵ value to determine the clusters. But, if the clusters have varying densities, this could lead to suboptimal partitioning of the data. HDBSCAN addresses this problem and we describe it next.

HDBSCAN

HDBSCAN (Hierarchical Density-Based Spatial Clustering of Applications with Noise) is an extension of DBSCAN (Section 3.2.3). As opposed to DBSCAN, HDBSCAN relies on different distance thresholds ϵ for each cluster, thus obtaining clusters of varying densities.

HDBSCAN first builds a hierarchy of clusters, based on various ϵ values selected in decreasing order. Then, based on such a hierarchy, HDBSCAN selects as final clusters the most persistent ones, where cluster persistence represents how long a cluster remains the same without splitting when decreasing the value of ϵ. In HDBSCAN, one has only to specify one parameter, which is the minimum number of individuals required to form a cluster, denoted by n [141]. Clusters with less than n individuals are considered noise and ignored.

3.2.4 Many Objective Optimization

Engineers are often faced with problems requiring to fulfill multiple objectives at the same time, called *multi-objective problems*. For instance, multi-objective search algorithms were used in test suite minimization approaches to balance cost, effectiveness, and other objectives [236, 252]. Multi-objective problems with at least (three or) four objectives are informally known as *many-objective problems* [122]. In both kind of problems, one needs a solution which is a good trade-off between the objectives. Hence, we first introduce the Pareto front of a decision space (Section 3.2.4). Then, we describe genetic algorithms able to solve many-objective problems (Section 3.2.4).

Pareto Front

Multi- and many-objective problems can be stated as minimizing several objective functions while taking values in a given decision space. The goal of multi-objective optimization is to approximate the Pareto Front in the objective space [122].

Formally, if D is the decision space and $f_1(.), \ldots, f_n(.)$ are n objective functions defined on D, then the *fitness vector* of a decision vector $x \in D$ is $[f_1(x), \ldots, f_n(x)]$, hereafter denoted $F(x)$. Moreover, a decision vector x_1 *Pareto-dominates* a decision vector x_2 (hereafter denoted $x_1 \succ x_2$) if 1) for each $1 \leq i \leq n$, we have $f_i(x_1) \leq f_i(x_2)$, and 2) there exists $1 \leq i \leq n$ such that $f_i(x_1) < f_i(x_2)$. If there exists no decision vector x_1 such that $x_1 \succ x_2$, we say that the decision vector x_2 is *non-dominated*. The *Pareto front* of D is the set $\{F(x_2) \mid x_2 \in D \text{ and } \forall x_1 \in D : x_1 \not\succ x_2\}$ of the fitness vectors of the non-dominated decision vectors. Finally, a *multi/many-objective problem* consists in:

$$\underset{x \in D}{\text{minimize}}\, F(x) = [f_1(x), \ldots, f_n(x)]$$

where the minimize notation means that we want to find or at least approximate the non-dominated decision vectors, hence the ones having a fitness vector in the Pareto front [122].

Solving Many-Objective Problems

Multi-objective algorithms like NSGA-II [72] or SPEA2 [108,259] are not effective in solving many-objective problems [73, 190] because of the following challenges:

1. The proportion of non-dominated solutions becomes exponentially large with an increased number of objectives. This reduces the chances of the search being stuck at a local optimum and may lead to a better convergence rate [122], but also slows down the search process considerably [73].

2. With an increased number of objectives, diversity operators (e.g., based on crowding distance or clustering) become computationally expensive [73].

3. If only a handful of solutions are to be found in a large-dimensional space, solutions are likely to be widely distant from each other. Hence, two distant parent solutions are likely to produce offspring solutions that are distant from them. In this situation, recombination operations may be inefficient and require crossover restriction or other schemes [73].

To tackle these challenges, several many-objective algorithms have been successfully applied within the software engineering community, like NSGA-III [73,98] and MOSA [190].

NSGA-III [73, 98] is based on NSGA-II [72] and addresses these challenges by assuming a set of supplied or predefined reference points. Diversity (challenge 2) is ensured by starting the search in parallel from each of the reference points, assuming that largely spread starting points would lead to exploring all relevant parts of the Pareto front. For each parallel search, parents share the same starting point, so they are assumed to be close enough so that recombination operations (challenge 3) are more meaningful. Finally, instead of considering all solutions in the Pareto front, NSGA-III focuses on individuals which are the closest to the largest number of reference points. That way, NSGA-III considers only a small proportion of the Pareto front (addressing challenge 1).

Another many-objective algorithm, MOSA [190], does not aim to identify a single individual achieving a trade-off between objectives but a set of individuals, each satisfying one of the objectives. Such characteristic makes MOSA adequate for many software testing problems where it is sufficient to identify one test case (i.e., an individual) for each test objective (e.g., covering a specific branch or violating a safety requirement). To deal with challenge 1, MOSA relies on a preference criterion amongst individuals in the Pareto front, by focusing on 1) extreme individuals (i.e., test cases having one or more objective scores equal to zero), and 2) in case of tie, the shortest test cases. These best extreme individuals are stored in an archive during the search, and the archive obtained at the last generation is the final solution. Challenges 2 and 3 are addressed by focusing the search, on each generation, on the objectives not yet covered by individuals in the archive.

3.3 Problem Definition

As the time required to execute a set of considered MRs may be large (Section 3.2.1), we aim to minimize the set of source inputs (hereafter, *input set*) to be used when applying MST to a Web system, given a set of MRs. In our context, each input is a sequence of actions used to communicate with the Web system and each action leads to a different Web page.

To ensure that a *minimized input set* can exercise the same vulnerabilities as the original one, intuitively, we should ensure that they belong to the same *input blocks*. Indeed, in software testing, after identifying an important characteristic to consider for the inputs, one can partition the input space in blocks, i.e., pairwise disjoint sets of inputs, such that inputs in the same block exercise the SUT in a similar way [31]. As the manual identification of relevant input blocks for a large system is extremely costly, we rely on clustering for that purpose (Section 3.6). Since an input is a sequence of actions, it can exercise several input blocks. In the rest of the thesis, we rely on the notion of *input coverage*, indicating the input blocks an input belongs to.

3.3.1 Assumptions and Goals

We assume we know, for each input in the initial input set, 1) its cost and 2) its coverage.

1) Because we want to make MST scalable, the cost $cost(in)$ of an input in corresponds to the execution time required to verify if the considered MRs are satisfied with this input. Because we aim to reduce this execution time without having to execute the MRs, as it would defeat the purpose of input set minimization, we use the number $nbActions(mr, in)$ of actions to be executed by an MR mr using input in as a surrogate metric for its execution time (see Section 3.2.1). We thus define the cost of an input as follows:

$$cost(in) \stackrel{\text{def}}{=} \sum_{mr \in MRs} nbActions(mr, in)$$

When $cost(in) = 0$, input in was not exercised by any MR due to the preconditions in these MRs. Hence, in is not useful for MST and can be removed without loss from the initial input set. Finally, the total cost of an input set I is $cost(I) \stackrel{\text{def}}{=} \sum_{in \in I} cost(in)$.

2) To minimize the cost of metamorphic testing, we remove unnecessary inputs from the initial input set, but we want to preserve all the inputs able to exercise distinct vulnerabilities. Hence, we consider, for each initial input in, its coverage $Cover(in)$. In our study, $Cover(in)$ is the set of input blocks the input in belongs to, and we determine these input blocks in Section 3.6 using *double-clustering*. For now, we assume that the coverage of an input is known. The total coverage of an input set I is $Cover(I) \stackrel{\text{def}}{=} \cup_{in \in I} Cover(in)$.

We can now state our goals. We want to obtain a subset $I_{final} \subseteq I_{init}$ of the initial input set such that 1) I_{final} does not reduce total input coverage, i.e., $Cover(I_{final}) = Cover(I_{init})$ and 2) I_{final} has minimal cost, i.e., $cost(I_{final}) = \min\{cost(I) \mid I \subseteq I_{init} \wedge Cover(I) = Cover(I_{init})\}$. Note that a solution I_{final} may not be necessarily unique.

3.3.2 A Many-Objective Problem

To minimize the initial input set, we focus on the *selection* of inputs that belong to the same input blocks as the initial input set. A potential solution to our problem is an input set $I \subseteq I_{init}$. Obtaining a solution I able to reach full input coverage is straightforward since, for each block bl, one can simply select an input in $Inputs(bl) \stackrel{\text{def}}{=} \{in \in I_{init} \mid bl \in Cover(in)\}$. The hard part of our problem is to determine a combination of inputs able to reach full input coverage at a minimal cost. Hence, we have to consider an input set as a whole and not focus on individual inputs.

This is similar to the *whole suite* approach [87] targeting white-box testing. They use as objective the total number of covered branches. But, in our context, counting the number of uncovered blocks would consider as equivalent input sets that miss the same number of blocks, without taking into account that it may be easier to cover some blocks than others (e.g., some blocks may be covered by many inputs, but some only by a few) or that a block may be covered by inputs with different costs. Thus, to obtain a combination of inputs that minimizes cost while preserving input coverage, we have to investigate how input sets cover each input block.

Hence, we are interested in covering each input block as an individual objective, in a way similar to the coverage of each code branch for white-box testing [190]. Because the total number of blocks to be covered is typically large (≥ 4), we deal with a *many-objective problem* [122]. This can be an advantage, because a many-objective reformulation of complex problems can reduce the probability of being trapped in local optima and may lead to a better convergence rate [190]. But this raises several challenges (Section 3.2.4) that we tackle while presenting our search algorithm (Section 3.8).

3.3.3 Objective Functions

To provide effective guidance to a search algorithm, we need to quantify when an input set is closer to the objective of covering a particular block bl than another input set. In other

words, if I_1 and I_2 are two input sets which do not cover bl but have the same cost, we need to determine which one is more desirable to achieve the goals introduced in Section 3.3.1 by defining appropriate objective functions.

In general, adding an input to an input set would not only cover bl, but would also likely cover other blocks, that would then be covered by several inputs, thus introducing the possibility to remove some of them without affecting coverage. To track how a given block bl is covered by inputs from a given input set I, we introduce the concept of *superposition* as $superpos(bl, I) \stackrel{\text{def}}{=} |Inputs(bl) \cap I|$. For instance, if $superpos(bl, I) = 1$, then there is only one input in I covering bl. In that case, this input is necessary to maintain the coverage of I. More generally, with the *redundancy* metric, we quantify how much an input in is necessary to ensure the coverage of an input set I it belongs to: $redundancy(in, I) \stackrel{\text{def}}{=} \min\{superpos(bl, I) \mid bl \in Cover(in)\} - 1$. The -1 is used to normalize the redundancy metric so that its range starts at 0. If $redundancy(in, I) = 0$, we say that in is *necessary* in I, otherwise we say that in is *redundant*. In the following, we denote $Redundant(I) \stackrel{\text{def}}{=} \{in \in I \mid redundancy(in, I) > 0\}$ the set of the redundant inputs in I.

To focus on the least costly input sets during the search (Section 3.8), we quantify the gain obtained by removing redundant inputs. If I contains a redundant input in, then we call *removal step* a transition from I to $I \setminus \{in\}$. Otherwise, we say that I is already *reduced*. Unfortunately, given two redundant inputs in_1 and in_2, removing in_1 may render in_2 necessary. Hence, when considering potential removal steps (e.g., removing either in_1 or in_2), one has to consider the order of these steps. We represent a *valid order* of removal steps by a sequence of inputs $[in_1, \ldots, in_n]$ to be removed from I such that, for each $0 \leq i < n$, in_{i+1} is redundant in $I \setminus \{in_1, \ldots, in_i\}$. We denote $ValidOrders(I)$ the set of valid orders of removal steps in I. Removing redundant inputs $in_1, \ldots, in_n$ leads to a reduction of cost $cost(in_1) + \cdots + cost(in_n)$. For each input set I, we consider the maximal *gain* from valid orders of removal steps:

$$gain(I) \stackrel{\text{def}}{=} \max \Big\{ \sum_{1 \leq i \leq n} cost(in_i) \;\Big|\; [in_1, \ldots, in_n] \in ValidOrders(I) \Big\}$$

To reduce the cost of computing this gain, we prove in the appendix (Theorem 1) that, to determine which orders of removal steps are valid, we can remove inputs in any arbitrary order, without having to resort to backtracking to previous inputs. Moreover, in our approach, we need to compute the gain only in situations when the number of redundant inputs is small (Section 3.8), thus exhaustively computing the gain is tractable.

Adding an input $in_1 \in Inputs(bl)$ to I would lead to a gain but would also result in additional cost, hence warranting we consider the benefit-cost balance $gain(I \cup \{in_1\}) - cost(in_1)$ to evaluate how efficiently bl is covered by in_1. More generally, we define the *potential* of I to efficiently cover bl as the maximum benefit-cost balance obtained by adding inputs in_1 in covering bl. But, as an input in_1 may be necessary to cover bl while leading to no or not enough removal steps, $gain(I \cup \{in_1\}) - cost(in_1)$ may be negative. To facilitate normalization, we need the potential to return a non-negative value, thus we shift all the benefit-cost balances for a given objective bl by adding a dedicated term. As the potential

is a maximum, the worst case of $gain(I \cup \{in_2\}) - cost(in_2)$ is when $gain(I \cup \{in_2\}) = 0$ and $cost(in_2)$ is the minimal cost amongst the inputs able to cover bl. Hence, we obtain the following definition for the potential of I in covering bl:

$$\begin{aligned} potential(I, bl) \stackrel{\text{def}}{=} & \max\{gain(I \cup \{in_1\}) - cost(in_1) \\ & \qquad \mid in_1 \in Inputs(bl)\} \\ & + \min\{cost(in_2) \mid in_2 \in Inputs(bl)\} \end{aligned}$$

We thus use the potential to define the objective function associated with an objective bl.

To normalize our metrics, we rely on the normalization function $\omega(x) \stackrel{\text{def}}{=} \frac{x}{x+1}$, which is used to reduce a range of values from $[0, \infty)$ to $[0, 1)$ while preserving the ordering. When used during a search, it is less prone to precision errors and more likely to drive faster convergence towards an adequate solution than alternatives [35]. We use it to normalize the cost between 0 and 1 and the smaller is the normalized cost, the better a solution is. For coverage, since a high potential is more desirable, we use its complement $\frac{1}{x+1} = 1 - \omega(x)$ to reverse the order, so that the more potential an input set has, the lower its coverage objective function is. We thus define the objective function corresponding to a block bl_i as:

$$f_{bl_i}(I) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } bl_i \in Cover(I) \\ \frac{1}{potential(I, bl_i)+1} & \text{otherwise} \end{cases}$$

The lower this value, the better. If bl_i is covered, then $f_{bl_i}(I) = 0$, otherwise $f_{bl_i}(I) > 0$. As expected, input sets that cover the objective are better than input sets that do not, and if two input sets do not cover the objective, then the normalized potential is used to break the tie.

3.3.4 Solutions to Our Problem

Each element in the decision space is an input set $I \subseteq I_{init}$, which is associated with a *fitness vector*:

$$F(I) \stackrel{\text{def}}{=} [\omega(cost(I)), f_{bl_1}(I), \ldots, f_{bl_n}(I)]$$

where $Cover(I_{init}) = \{bl_1, \ldots, bl_n\}$ denotes the n input blocks to be covered by input sets $I \subseteq I_{init}$.

Hence, we can define the Pareto front formed by the non-dominated solutions in our decision space (Section 3.2.4) and we formulate our problem definition as a many-objective optimization problem:

$$\underset{I \subseteq I_{init}}{\text{minimize}}\, F(I)$$

where the minimize notation means that we want to find or at least approximate the non-dominated decision vectors having a fitness vector on the Pareto front [122]. Because we want full input coverage (Section 3.3.1), the ultimate goal is a non-dominated solution I_{final} such that:

$$F(I_{final}) = [\omega(cost_{\min}), 0, \ldots, 0]$$

where $cost_{\min}$ is the cost of the cheapest subset of I_{init} with full input coverage.

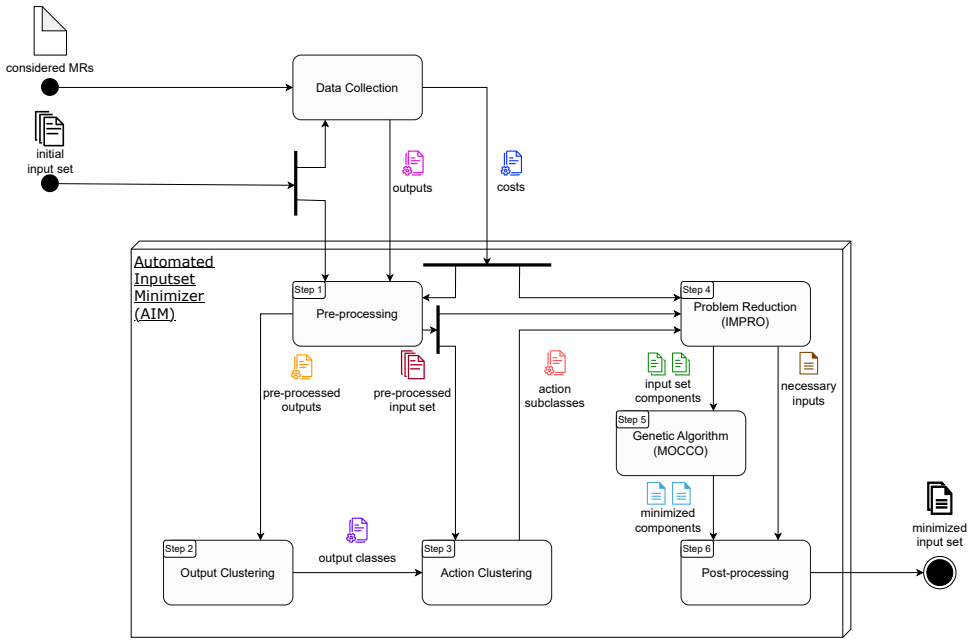

Figure 3.2: Activity diagram of the Automated Input set Minimizer (AIM) approach.

3.4 Overview of the approach

As stated in our problem definition (Section 3.3), we aim to reduce the cost of MST (Section 3.2.1) by minimizing an initial input set without removing inputs that are required to exercise distinct security vulnerabilities. To do so, we need to tackle the following sub-problems (Section 3.3.4):

1. For each initial input, we need to determine its cost without executing the considered MRs.

2. For each initial input, we need to determine its coverage. In the context of metamorphic testing for Web systems, we consider input blocks based on system outputs and input parameters.

3. Amongst all potential input sets $I \subseteq I_{init}$, we search for a non-dominated solution I_{final} that preserves coverage while minimizing cost.

The *Automated Input set Minimizer (AIM)* approach relies on analyzing the output and cost corresponding to each input. AIM obtains such information through a new feature added to the *MST-wi* toolset to execute each input on the system and retrieve the content of the corresponding Web pages. Obtaining the outputs of the system is very inexpensive compared to executing the considered MRs. Moreover, to address our first sub-problem, we also updated *MST-wi* to retrieve the cost of an input without executing the considered MRs. We rely on a surrogate metric (Section 3.2.1), linearly correlated with execution time, which is inexpensive to collect (Section 3.5.1).

In Step 1 (Pre-processing), AIM pre-processes the initial input set and the output information, by extracting relevant textual content from each returned Web page (Section 3.5.2).

To address the second sub-problem, AIM relies on a *double-clustering* approach (Section 3.6), which is implemented by Step 2 (Output Clustering) and Step 3 (Action Clustering). For both steps, AIM relies on state-of-the-art clustering algorithms, which require to select hyper-parameter values (Section 3.6.1). *Output clustering* (Section 3.6.2) is performed on the pre-processed outputs, each generated cluster corresponding to an *output class*. Then, for each output class identified by the Output Clustering Step, *Action clustering* (Section 3.6.3) first determines the actions whose output belongs to the considered output class, then partitions these actions based on action parameters such as URL, username, and password, obtaining *action subclasses*. On the completion of Step 3, AIM has mapped each input to a set of action subclasses, used for measuring input coverage as per our problem definition (Section 3.3.1).

To preserve diversity in our input set, and especially to retain inputs that are necessary to exercise vulnerabilities, we require the minimized input set generated by AIM to cover the same action subclasses as the initial input set. That way, we increase the chances that the minimized input set contains at least one input able to exercise each vulnerability detectable with the initial input set.

Using cost and coverage information, AIM can address the last sub-problem. Since the size of the search space exponentially grows with the number of initial inputs, the solution cannot be obtained by exhaustive search. Actually, our problem is analogous to the knapsack problem [123], which is NP-hard, and is thus unlikely to be solved by deterministic algorithms. Therefore, AIM relies on meta-heuristic search to find a solution (Step 5) after reducing the search space (Step 4).

In Step 4, since the search space might be large, AIM first reduces the search space to the maximal extent possible (Section 3.7) before resorting to meta-heuristic search. Precisely, it relies on the Inputset Minimization Problem Reduction Operator (IMPRO) component for problem reduction, which determines the necessary inputs, removes inputs that cannot be part of the solution, and partitions the remaining inputs into input set *components* that can be independently minimized.

In Step 5, AIM applies a genetic algorithm (Section 3.8) to minimize each component. Because existing algorithms did not entirely fit our needs, we explain why we introduce MOCCO (Many-Objective Coverage and Cost Optimizer), a novel genetic algorithm which converges towards a solution covering the objectives at a minimal cost, obtaining a *minimized input set component*.

Finally, after the genetic search is completed for each component, in Step 6 (Post-processing), AIM generates the *minimized input set* by combining the necessary inputs identified by IMPRO and the inputs from the MOCCO minimized components (Section 3.9).

Note that, even though in our study we focus on Web systems, steps 4 (IMPRO), 5 (MOCCO), and 6 (post-processing), which form the core of our solution, are generic and

can be applied to any system. Moreover, step 1 (pre-processing) and steps 2 and 3 (double-clustering) can be tailored to apply AIM to other domains (e.g., desktop applications, embedded systems). And though we relied on *MST-wi* to collect our data, AIM does not depend on a particular data collector, and using or implementing another data collector would enable the use of our approach in other contexts.

3.5 Step 1: Data Collection and Pre-Processing

In Step 1, AIM determines the cost of each initial input (Section 3.5.1) and extracts meaningful textual content from the Web pages obtained with the initial inputs (Section 3.5.2).

3.5.1 Input Cost

The cost of a source input is the number of actions executed by source and follow-up inputs for the considered MRs (Section 3.3.1). Note that counting the number of actions to be executed is inexpensive compared to executing them on the SUT, then checking for the verdict of the output relation. For instance, counting the number of actions for eleven MRs with Jenkins' initial input set (Section 3.10) took less than five minutes, while executing the MRs took days.

3.5.2 Output Representation

Since, in this study, we focus on Web systems, the outputs of the SUT are Web pages. Fortunately, collecting these pages using a crawler and extracting their textual content is inexpensive compared to executing MRs. Hence, we can use system outputs to determine relevant input blocks (Section 3.3).

We focus on textual content extracted from the Web pages returned by the Web system under test. We remove from the textual content of each Web page all the data that is shared among many Web pages and thus cannot characterize a specific page, like system version, date, or (when present) the menu of the Web page. Moreover, to focus on the meaning of the Web page, we consider the remaining textual content not as a string of characters but as a sequence of words. Also, following standard practice in natural language processing, we apply a stemming algorithm to consider distinct words with the same stem as equivalent, for instance the singular and plural forms of the same word. Finally, we remove stopwords, numbers, and special characters, in order to focus on essential textual information.

3.6 Steps 2 and 3: Double Clustering

To reduce the cost of MST, we want to minimize an initial input set while preserving, for each vulnerability affecting the SUT, at least one input able to exercise it; of course, in

practice, such vulnerabilities are not known in advance but should be discovered by MST. Hence, we have to determine in which cases two inputs are distinct enough so that both should be kept in the minimized input set, and in which cases some inputs are redundant with the ones we already selected and thus can be removed. To determine which inputs are similar and which significantly differ, we rely on clustering algorithms. Precisely, we rely on the K-means, DBSCAN, and HDBSCAN algorithms to cluster our data points. Each of them has a set of hyper-parameters to be set and we first detail how these hyper-parameters are obtained using *Silhouette analysis* (Section 3.6.1).

Since, for practical reasons, we want to avoid making assumptions regarding the nature of the Web system under test (e.g., programming language or underlying middleware), we propose a black-box approach relying on input and output information to determine which inputs we have to keep or remove. In the context of a Web system, each input is a sequence of actions, each action enables a user to access a Web page (using a POST or GET request method), and each output is a Web page. After gathering output and action information, we perform *double-clustering* on our data points, i.e., two clustering steps performed in sequence:

1. *Output clustering* (Section 3.6.2) uses the outputs of the Web system under test, i.e., textual data obtained by pre-processing content from Web pages (Section 3.5.2). We define an output distance (Section 3.6.2) to quantify similarity between these outputs, which is then used to run Silhouette analysis and clustering algorithms to partition outputs into *output classes* (Section 3.6.2).

2. *Action clustering* (Section 3.6.3) then determines input coverage. First, AIM collects in the same *action set* actions leading to outputs in the same output class (Section 3.6.3). Then, AIM refines each action set by partitioning the actions it contains using action parameters. To do so, it first uses the request method (Section 3.6.3) to split action sets into parts. Then, we define an action distance (Section 3.6.3) based on the URL (Section 3.6.3) and other parameters (Section 3.6.3) of the considered actions. Finally, AIM relies on Silhouette analysis and clustering algorithms to partition each part of an action set into *action subclasses* (Section 3.6.3), defining our input blocks (Section 3.3.1).

Note that *double-clustering* should not be confused with *biclustering* [54,192], since the latter simultaneously clusters two distinct aspects (features and samples) of the data, while the former clusters only one aspect (actions, in our case, that can be seen as features) but in two consecutive steps (action outputs, then action parameters), the second refining the first one.

3.6.1 Hyper-parameters Selection

In this study, we rely on the common K-means [37], DBSCAN [82], and HDBSCAN [141] clustering algorithms (Section 3.2.3) to determine output classes (Section 3.6.2) and action subclasses (Section 3.6.3). These clustering algorithms require a few hyper-parameters to

be set. One needs to select for K-means the number of clusters k, for DBSCAN the distance threshold ϵ and the minimum number of neighbors n, and for HDBSCAN the minimum number n of individuals required to form a cluster.

To select the best values for these hyper-parameters, we rely on *Silhouette analysis*. Though the Silhouette score is a common metric used to determine optimal values for hyper-parameters [39,40], it is obtained from the average Silhouette score of the considered data points. Thus, for instance, clusters with all data points having a medium Silhouette score cannot be distinguished from clusters where some data points have a very large Silhouette score while others have a very small one. Hence, having a large Silhouette score does not guarantee that all the data points are well-matched to their cluster. To quantify the variability in the distribution of Silhouette scores, we use Gini index, a common measure of statistical dispersion. If the Gini index is close to 0, then Silhouette scores are almost equal. Conversely, if it is close to 1, then the variability in Silhouette score across data points is large.

Hence, for our Silhouette analysis, we consider two objectives: (average) Silhouette score and the Gini index of the Silhouette scores. The selection of hyper-parameters is therefore a multi-objective problem with two objectives. We rely on the common NSGA-II evolutionary algorithm [72] to solve this problem and approximate the Pareto front regarding both Silhouette score and Gini index. Then, we select the item in the Pareto front that has the highest Silhouette score.

3.6.2 Step 2: Output Clustering

Output clustering consists in defining an output distance (Section 3.6.2) to quantify dissimilarities between Web system outputs, and then to partition the outputs to obtain *output classes* (Section 3.6.2).

A user communicates with a Web system using actions. Hence, an input for a Web system is a sequence of actions (e.g., login, access to a Web page, logout). As the same action may occur several times in an input in, a given occurrence of an action is identified by its position i in the input and denoted $action(in, i)$. Outputs of a Web system are textual data obtained by pre-processing the content from Web pages (Section 3.5.2). The accessed Web page depends not only on the considered action, but also on the previous ones; for instance if the user has logged into the system. Hence, we denote by $output(in, i)$ the output of the action at position i in in.

Output distance

In this study, we use system outputs (i.e., Web pages) to characterize system states. Hence, two actions that do not lead to the same output should be considered distinct because they bring the system into different states. More generally, dissimilarity between outputs is quantified using an *output distance*. Since we deal with textual data, we consider both Levenshtein and bag distances. Levenshtein distance is usually a good representation of the

difference between two textual contents [95,228]. However, computing the minimal number of edits between two strings can be costly, since the complexity of the Levenshtein distance between two strings is $O(len(s_1) \times len(s_2))$, where $len(.)$ is the length of the string [254]. Thus, we consider the bag distance [142] as an alternative to the Levenshtein distance, because its complexity is only $O(len(s1) + len(s_2))$ [44]. But it does not take into account the order of words and is thus less precise than Levenshtein distance.

Output Classes

We partition the textual content we obtained from Web pages (Section 3.5.2) using the K-means, DBSCAN, and HDBSCAN clustering algorithms, setting the hyper-parameters using Silhouette analysis (Section 3.6.1), and determining similarities between outputs using the chosen output distance. We call *output classes* the obtained clusters and we denote by $OutputClass(in, i)$ the unique output class $output(in, i)$ belongs to.

3.6.3 Step 3: Action Clustering

Exercising all the Web pages is not sufficient to discover all the vulnerabilities; indeed, vulnerabilities might be detected through specific combinations of parameter values associated to an action (e.g., values belonging to a submitted form). Precisely, actions on a Web system can differ with respect to a number of *parameters* that include the URL (allowing the action to perform a request to a Web server), the method of sending a request to the server (like GET or POST), URL parameters (e.g., `http://myDomain.com/myPage?urlParameter1=value1&urlParameter2=value2`), and entries in form inputs (i.e., textarea, textbox, options in select items, datalists).

Based on the obtained output classes, *action clustering* first determines *action sets* (Section 3.6.3). Then, action clustering refines each action set by partitioning the actions it contains using actions parameters. First, we give priority to the method used to send a request to the server, so we split each action set using the request method (Section 3.6.3). Then, to quantify the dissimilarity between two actions, we define an action distance (Section 3.6.3) based on URL (Section 3.6.3) and other parameters (Section 3.6.3). That way, action clustering refines each action set into *action subclasses* (Section 3.6.3).

Action Sets

Based on the obtained output classes (Section 3.6.2), AIM determines *action sets* such that actions leading to outputs in the same output class *outCl* are in the same action set:

$$ActionSet(outCl) \stackrel{\text{def}}{=} \{act \mid \exists in, i : action(in, i) = act \\ \wedge\ OutputClass(in, i) = outCl\}$$

Note that, because an action can have different outputs depending on the considered input, it is possible for an action to belong to several actions sets, corresponding to several output classes.

Request Partition

Each action uses either a POST or GET method to send an HTTP request to the server. Actions (such as login) that send the parameters to the server in the message body use the POST method, while actions that send the parameters through the URL use the GET method. As this difference is meaningful for distinguishing different action types, we split each action set into two parts: the actions using a POST method and those using a GET method.

Action Distance

After request partition (Section 3.6.3), we consider one part of an action set at a time and we refine it using an action distance quantifying dissimilarity between actions based on remaining parameters (e.g., URL or form entries). In the context of a Web system, each Web content is identified by its URL, so we give more importance to this parameter. We denote $url(act_i)$ the URL of action act_i. For the sake of clarity, we call in the rest of the section *residual parameters* the parameters of an action which are not its request method nor its URL and we denote $res(act_i)$ the residual parameters of action act_i. Since we give more importance to the URL, we represent the distance between two actions by a real value, where the integral part corresponds to the distance between their respective URLs and the decimal part to the distance between their respective residual parameters:

$$\begin{aligned} actionDist(act_1, act_2) \stackrel{\text{def}}{=} & \, urlDist(url(act_1), url(act_2)) \\ & + paramDist(res(act_1), res(act_2)) \end{aligned}$$

where the URL distance $urlDist(.,.)$ is defined in Section 3.6.3 and returns an integer, and the parameter distance $paramDist(.,.)$ is defined in Section 3.6.3 and returns a real number between 0 and 1.

URL distance

A URL is represented as a sequence of at least two words, separated by '://' between the first and second word, then by '/' between any other words. The length of a URL url is its number of words, denoted $len(url)$. Given two URLs, url_1 and url_2, their *lowest common ancestor* is the longest prefix they have in common, denoted $LCA(url_1, url_2)$. We define the distance between two URLs as the total number of words separating them from their lowest common ancestor:

$$\begin{aligned} urlDist(url_1, url_2) \stackrel{\text{def}}{=} & len(url_1) + len(url_2) \\ & - 2 \times len(LCA(url_1, url_2)) \end{aligned}$$

We provide an example in Figure 3.3.

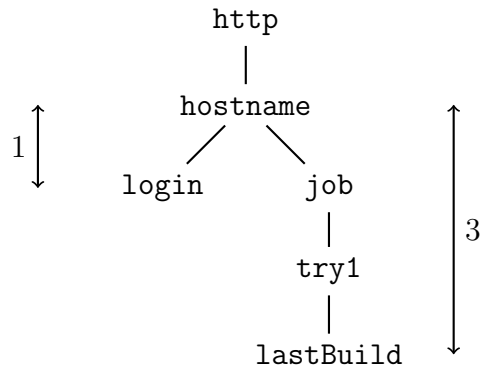

Figure 3.3: The URL distance between `http://hostname/login` and `http://hostname/job/try1/lastBuild` is $1 + 3 = 4$.

Parameter Distance

To quantify the dissimilarity between residual parameters, we first independently quantify the dissimilarity between pairs of parameters of the same type. Since, in our context, we exercise vulnerabilities by only using string or numerical values, we ignore parameter values of other types such as byte arrays (e.g., an image uploaded to the SUT). In other contexts, new parameter distance functions suited to other input types may be required. For strings, we use the Levenshtein distance [95, 228], whereas for numerical values we consider the absolute value of their difference [51]:

$$paramValDist(v_1, v_2) \stackrel{\text{def}}{=} \begin{cases} LevenshteinDist(v_1, v_2) & \text{if } type(v_1) = \text{str} = type(v_2) \\ |v_1 - v_2| & \text{if } type(v_1) = \text{int} = type(v_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Since we have parameters of different types, we normalize the parameter distance using the normalization function $\omega(x) = \frac{x}{x+1}$ (Section 3.3.3). Then, we add these normalized distances together, and normalize the sum to obtain a result between 0 and 1. We compute the parameter distance in case of *matching parameters*, i.e., the number of parameters is the same and the corresponding parameters have the same type. Otherwise, we assume the largest distance possible, which is 1 due to the normalization. This is the only case where the value 1 is reached, as distance lies otherwise in [0 1[, as expected for a decimal part (Section 3.6.3):

$$paramDist(resids_1, resids_2) \stackrel{\text{def}}{=} \begin{cases} \omega\left(\sum\limits_{0 \le i < len(resids_1)} \omega(paramValDist(resids_1^{[i]}, resids_2^{[i]}))\right) \\ \quad \text{if } resids_1 \text{ and } resids_2 \text{ have matching parameters} \\ 1 \text{ otherwise} \end{cases}$$

where $resids_1 = res(act_1)$, $resids_2 = res(act_2)$, and $resids^{[i]}$ is the i-th element of $resids$.

For instance, we consider two actions act_1 and act_2 having matching parameters with the values in Table 3.1 for page number, username, and password. The distance for the

Table 3.1: Values for the Example of Parameter Distance

	Page Number	Username	Password
$resids_1$	10	"John"	"qwerty"
$resids_2$	42	"Johnny"	"qwertyuiop"

page number is $paramValDist(10, 42) = 32$, normalized into $\frac{32}{32+1} \approx 0.97$. For the username, it is $paramValDist(\text{"John"}, \text{"Johnny"}) = 2$, normalized into $\frac{2}{2+1} \approx 0.66$. For the password, it is $paramValDist(\text{"qwerty"}, \text{"qwertyuiop"}) = 4$, normalized into $\frac{4}{4+1} = 0.80$. Thus, the parameter distance is $paramDist(resids_1, resids_2) \approx \frac{0.97+0.66+0.80}{0.97+0.66+0.80+1} \approx 0.71$.

Action Subclasses

We partition both parts of each action set (Section 3.6.3) using the K-means, DBSCAN, or HDBSCAN clustering algorithms, setting the hyper-parameters using our Silhouette analysis (Section 3.6.1), and quantifying action dissimilarity using our action distance (Section 3.6.3), obtaining clusters we call *action subclasses*. We denote by $ActionSubclass(act, actSet)$ the unique action subclass bl from the action set $actSet$ such that $act \in bl$. For the sake of simplicity, we denote $Subclass(in, i)$ the action subclass corresponding to the i-th action in input in:

$$Subclass(in, i) \stackrel{\text{def}}{=} ActionSubclass(action(in, i), \\ ActionSet(OutputClass(in, i))$$

Finally, in our study, the objectives covered by an input are:

$$Cover(in) \stackrel{\text{def}}{=} \{Subclass(in, i) \mid 1 \leq i \leq len(in)\}$$

3.7 Step 4: Problem Reduction

The search space for our problem (Section 3.3.4) consists of all the subsets of the initial input set, which leads to 2^m potential solutions, where m is the number of initial inputs.

For this reason, AIM integrates a *problem reduction* step, implemented by the Inputset Minimization Problem Reduction Operator (IMPRO) component, to minimize the search space before solving the search problem in the next step (Section 3.8). We apply the following techniques to reduce the size of the search space:

- *Determining redundancy*: Necessary inputs (Section 3.3.3) must be part of the solution, hence one can only investigate redundant inputs (Section 3.7.3). Moreover, one can restrict the search by removing the objectives already covered by necessary inputs. Finally, if a redundant input does not cover any of the remaining objectives, it will not contribute to the final coverage, and hence can be removed.
- *Removing duplicates*: Several inputs may have the same cost and coverage. In this case, we consider them as *duplicates* (Section 3.7.4). Thus, we keep only one and we remove the others.

- *Removing locally-dominated inputs*: For each input, if there exists other inputs that cover the same objectives at a same or lower cost, then the considered input is *locally-dominated* by the other inputs (Section 3.7.5) and is removed.
- *Dividing the problem*: We consider two inputs covering a common objective as being connected. Using this relation, we partition the search space into connected *components* that can be independently solved (Section 3.7.6), thus reducing the number of objectives and inputs to investigate at a time.

Before detailing these techniques, we first explain in which order there are applied (Section 3.7.1).

3.7.1 Order of Reduction Techniques

We want to perform first the least expensive reduction techniques, to sequentially reduce the cost of the following more expensive techniques. Determining redundancy requires $\mathcal{O}(m \times c)$ steps, removing duplicates requires $\mathcal{O}(m^2)$ steps, and removing locally-dominated inputs requires $\mathcal{O}(m \times 2^n)$ steps, where m is the number of inputs, c is the maximal number of objectives covered by an input, and n is the maximal number of neighbors for an input (i.e., the number of other inputs that cover an objective shared with the considered input). In our study, we assume $c < m$. Hence, we first determine redundancy, then remove duplicates, and remove locally-dominated inputs. Dividing the problem requires exploring neighbors and comparing non-visited inputs with visited ones, so it is potentially the most costly of these reduction techniques; hence, it is performed at the end.

After determining redundancy, the removal of already covered objectives may lead to new inputs being duplicates or locally-dominated. Moreover, the removal of duplicates or locally-dominated inputs may lead to changes in redundancy, making some previously redundant inputs necessary. Hence, these reduction techniques should be iteratively applied, until a stable output is reached. Such output can be detected by checking if inputs were removed during an iteration.

Therefore, the order is as follows. We first initialize variables (Section 3.7.2). Then we repeat, until no input is removed, the following steps: determine redundancy (Section 3.7.3), remove duplicates (Section 3.7.4), and remove locally-dominated inputs (Section 3.7.5). Finally, we divide the problem into sub-problems (Section 3.7.6).

3.7.2 Initializing Variables

During problem reduction, we consider three variables: I_{necess}, the set of the inputs that has to be part of the final solution, I_{search}, the remaining inputs to be investigated, and $Coverage_{obj}$, the objectives that remain to be covered by subsets of I_{search}. I_{necess} is initially empty. I_{search} is initialized as the initial input set. $Coverage_{obj}$ is initialized as the coverage of the initial input set (Sections 3.3.1 and 3.6.3).

3.7.3 Determining Redundancy

Algorithm 1 Redundancy determination technique.

1: **procedure** REDUNDANCY($I_{necess}, I_{search}, Coverage_{obj}$)
2: $I_{redund} \leftarrow Redundant(I_{search})$
3: $I^{new}_{necess} \leftarrow I_{search} \setminus I_{redund}$
4: $I_{necess} \leftarrow I_{necess} \cup I^{new}_{necess}$
5: $Coverage_{obj} \leftarrow Coverage_{obj} \setminus Cover(I^{new}_{necess})$
6: $I_{search} \leftarrow \{in \in I_{redund} |\ Cover(in) \cap Coverage_{obj} \neq \varnothing\}$
7: **return** $I_{necess}, I_{search}, Coverage_{obj}$

This technique is presented in Algorithm 1. Each time it is repeated, the redundancy of the remaining inputs is computed (Line 2). Among them, inputs which are necessary (Section 3.3.3) in I_{search} (Line 3) for the objectives in $Coverage_{obj}$ have to be included in the final solution (Line 4), otherwise some objectives will not be covered. Then, the objectives already covered by the necessary inputs are removed (Line 5). Hence, in the following, we only consider, for each remaining input $in \in I_{search}$, their coverage regarding the remaining objectives, i.e., $Cover(in) \cap Coverage_{obj}$, instead of $Cover(in)$.

Finally, some redundant inputs may cover only objectives that are already covered by necessary inputs. In that case, they cannot be part of the final solution because they would contribute to the cost but not to the coverage of the objectives. Hence, we restrict without loss the search space for our problem by considering only redundant inputs that can cover the remaining objectives (Line 6).

3.7.4 Removing Duplicates

In the many-objective problem described in Section 3.3.3, inputs are characterized by their coverage (Section 3.6.3) and their cost (Section 3.3.1). Hence, two inputs with the same coverage and cost are considered duplicates. In that case, IMPRO selects one and removes the other.

3.7.5 Removing Locally-dominated Inputs

For a given input $in \in I_{search}$, if the same coverage can be achieved by one or several other input(s) for at most the same cost, then in is not required for the solution. Formally, we say that the input $in \in I_{search}$ is *locally dominated* by the subset $S \subseteq I_{search}$, denoted $in \sqsubseteq S$, if $in \notin S$, $Cover(in) \subseteq Cover(S)$, and $cost(in) \geq cost(S)$. In order to simplify the problem, inputs that are locally dominated should be removed from the remaining inputs I_{search}.

Removing a redundant input in (Section 3.3.3) can only affect the redundancy of the inputs in I that cover objectives in $Cover(in)$. Hence, we consider two inputs as being

	bl_1	bl_2	bl_3	bl_4	bl_5	bl_6	bl_7
in_1	✓	✓					
in_2		✓	✓				
in_3			✓	✓			
in_4					✓	✓	
in_5						✓	✓

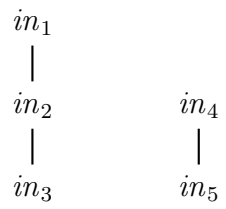

Figure 3.4: Inputs covering one objective in common (left) are connected in the corresponding overlapping graph (right).

connected if they cover at least one common objective. Formally, we say that two inputs in_1 and in_2 *overlap*, denoted by $in_1 \sqcap in_2$, if $Cover(in_1) \cap Cover(in_2) \neq \varnothing$.

One concern is that removing a locally-dominated input could alter the local dominance of other inputs. Fortunately, this is not the case for local dominance. For every locally-dominated input $in \in I_{search}$, there always exists a subset $S \subseteq I_{search}$ of not locally-dominated inputs such that $in \sqsubseteq S$. Hence, the locally dominated inputs can be removed in any order without reducing coverage or preventing cost reduction, both being ensured by non locally-dominated inputs. Therefore, IMPRO keeps in the search space only the remaining inputs that are not locally-dominated:

$$I_{search} \leftarrow \{in \in I_{search} \mid \forall S \subseteq I_{search} : in \not\sqsubseteq S\}$$

3.7.6 Dividing the Problem

After removing as many inputs as possible, we leverage the overlapping relation (Section 3.7.5) to partition the remaining inputs into connected components, in a divide-and-conquer approach. We denote $\mathcal{G}_{\sqcap}(I)$ the *overlapping graph* of the input set I, i.e., the undirected graph such that vertices are inputs in I and edges correspond to the overlapping relation $\sqcap$, and $Comps(I)$ the set of the *connected components* of $\mathcal{G}_{\sqcap}(I)$.

For instance, in Figure 3.4, we represent on the left the input blocks covered by each input in the search space and the corresponding overlapping graph on the right. The connected components of the graph are $\{in_1, in_2, in_3\}$ and $\{in_4, in_5\}$.

Such connected components are important because inputs in a connected component can be removed without altering the redundancy of inputs in other connected components. Similarly, the gain, i.e., the maximal cost reduction from removing redundant inputs (Section 3.3.3), can be independently computed on each connected component, i.e., for each input set I, $gain(I) = \sum_{C \in Comps(I)} gain(C)$.

Hence, instead of searching for a solution on I_{search} to solve our initial problem (Section 3.3.4), we use a divide-and-conquer strategy to split the problem into more manageable sub-problems that can be independently solved on each connected component $C \in Comps(I_{search})$.

We denote $Coverage_{obj}(C) \stackrel{\text{def}}{=} Coverage_{obj} \cap Cover(C)$ the remaining objectives to be covered by inputs in C and we formulate the sub-problem on the connected component similarly to the initial problem:

$$\underset{I \subseteq C}{\text{minimize}}\, F_C(I) \stackrel{\text{def}}{=} [\omega(cost(I)), f_{bl_1}(I), \ldots, f_{bl_n}(I)]$$

where $Coverage_{obj}(C) = \{bl_1, \ldots, bl_n\}$ and the minimize notation is detailed in Section 3.2.4. We denote I_C a non-dominated solution with full coverage, such that $F_C(I_C) = [\omega(cost_{\min}), 0, \ldots, 0]$.

3.8 Step 5: Genetic Search

Our initial goal of obtaining a subset $I_{final} \subseteq I_{init}$ with total input coverage at minimal cost (Section 3.3.1) can be expressed as a weighted set cover problem. Given a universe $\mathcal{U}$, a set $S = \{S_1, S_2, \ldots\}$ of subsets $S_i \subseteq \mathcal{U}$, and a weight function mapping each subset S_i to a positive real number, the weighted set cover problem consists in finding a subset $T \subseteq S$ such that subsets S_i in T cover $\mathcal{U}$ at a minimal cumulative cost. To express our initial goal as a weighted set cover problem, we consider as universe the input blocks $\mathcal{U} = Cover(I_{init})$ to cover, as subsets of the universe the input blocks $Cover(in)$ covered by each input $in \in I_{init}$, and as weight of a subset $Cover(in)$ the cost $cost(in)$ of the corresponding input. A solution to this set cover problem is a set of subsets $T = \{Cover(in_1), \ldots, Cover(in_n)\}$, providing a minimized input set $\{in_1, \ldots, in_n\}$, given that each subset $Cover(in)$ can be uniquely mapped to its initial input in. This is the case for each connected component C, since IMPRO removed duplicates (Section 3.7.4) and locally-dominated inputs (Section 3.7.5). While the set cover problem is NP-hard [112], it admits a very efficient [216] polynomial time approximation using a greedy algorithm, as well as various techniques to find approximate solutions to an ILP formulation of the problem [233]. Unfortunately, the set cover problem considers only the total coverage and the cumulative cost of solutions, not input blocks as individual objectives (Section 3.3.2) as in the formulation of our many-objective problem (Section 3.3.4) or sub-problem (Section 3.7.6). Hence, an algorithm solving the set cover problem may not take into account relevant information about how each input block is covered. Indeed, some blocks may be more difficult to cover and should thus receive priority. Further, an input selected by the greedy algorithm, because it covers many blocks at small cost, may be less optimal than several inputs covering more blocks at slightly larger cost. For instance, if $Cover(in_1) = \{bl_1, bl_2\}$ with $cost(in_1) = 2$, $Cover(in_2) = \{bl_1, bl_3\}$ with $cost(in_2) = 3$, and $Cover(in_3) = \{bl_2, bl_4\}$ with $cost(in_3) = 3$, the greedy algorithm selects in_1 because it is locally optimal, then in_2 and in_3 to cover respectively bl_3 and bl_4, at a total cost of 8. However, selecting only in_2 and in_3 leads to full coverage but at a lower cost of 6. In this example, the greedy algorithm selects in_1 covering blocks bl_1 and bl_2, which are easier to cover because each block can be covered by two inputs, while blocks bl_3 and bl_4 can only be covered by one input. Therefore, it is unlikely for a polynomial time approximation of the set cover problem to find, in general, the best solution to our many-objective problem.

Alternatively, obtaining an optimal solution I_C to our sub-problem on a connected component C is similar to solving the knapsack problem, which is also NP-hard [123]. To be more precise, our problem is equivalent to the 0-1 knapsack problem, which consists in selecting a subset of items to maximize a total value, while satisfying a weight capacity. Since we consider a many-objective problem (Section 3.3.2), we must address the multidimensional variant of the 0-1 knapsack problem, where each item has many "weights", one per considered objective. In our case, we minimize the total cost instead of maximizing total value and ensure the coverage of each action objective instead of making sure that each weight capacity is not exceeded. Furthermore, the 0-1 multidimensional knapsack problem is harder than the initial knapsack problem as it does not admit a fully polynomial time approximation scheme [111], hence the need for a meta-heuristic.

For our approach to scale, we adopt a genetic algorithm because it is known to find good approximations in reasonable execution time [190] and has been widely used in software testing. An input set $I \subseteq C$ can thus be seen as a chromosome, where each gene corresponds to an input $in \in C$, the gene value being 1 if $in \in I$ and 0 otherwise. Though several many-objective algorithms have been successfully applied within the software engineering community, like NSGA-III [73, 98] and MOSA [190] (Section 3.2.4), these algorithms do not entirely fit our needs (Section 3.8.1). Hence, we propose MOCCO (Many-Objective Coverage and Cost Optimizer), a novel genetic algorithm based on two populations and summarized in Algorithm 2. We first explain how these populations are initialized (Section 3.8.2). Then, for each generation, MOCCO performs the following standard steps: selection of the parents (Section 3.8.3), crossover of the parents to produce an offspring (Section 3.8.4), mutation of the offspring (Section 3.8.5), and update of the populations (Section 3.8.6) to obtain the next generation. The process continues until a termination criterion is met (Section 3.8.7). Then, we detail how MOCCO determines the solution I_C to each sub-problem.

3.8.1 Motivation for a Novel Genetic Algorithm

While our problem is similar to the multidimensional 0-1 knapsack problem, it is not exactly equivalent, since standard solutions to the multidimensional knapsack problem have to ensure that, for each "weight type", the total weight of the items in the knapsack is below weight capacity, while we want to ensure that, for each objective, at least one input in the minimized input set covers it. Hence, standard solutions based on genetic search are not applicable in our case, and we focus on genetic algorithms able to solve many-objective problems in the context of test case generation or minimization. We have explained earlier (Section 3.2.4) the challenges raised by many-objective problems and how the NSGA-III [73, 98] and MOSA [190] genetic algorithms tackle such challenges.

NSGA-III has the advantage of letting users choose the parts of the Pareto front they are interested in, by providing reference points. Otherwise, it relies on a systematic approach to place points on a normalized hyperplane. While this approach is useful in general, we are interested only in solutions that are close to a utopia point $[0, 0, \ldots, 0]$ (Section 3.3.3) covering every objective at no cost. Hence, we do not care about diversity over the Pareto

Algorithm 2 MOCCO overview.

```
1: procedure MOCCO(C, n_size, n_gens, time_budget)
2:     time_start ← getTime()
3:     Roofers ← initRoofers(C, n_size)
4:     Misers ← ∅
5:     n ← 1
6:     stillTime ← True
7:     while n ≤ n_gens ∧ stillTime do
8:         n ← n + 1
9:         I_1, I_2 ← selectParents(Roofers, Misers)
10:        I_3, I_4 ← crossover(I_1, I_2)
11:        for I ∈ {I_3, I_4} do
12:            I ← mutate(I)
13:            I ← reduce(I)
14:            if getTime() − time_start > time_budget then
15:                stillTime ← False
16:                break
17:            if I ∈ Roofers ∪ Misers then
18:                continue
19:            if Cover(I) = Coverage_obj(C) then
20:                Roofers ← updRoofers(Roofers, I)
21:            else
22:                Misers ← updMisers(Misers, I)
23:    I_C ← selectSolution(Roofers)
24:    return I_C
```

front, and we want to explore a very specific region of the search space. Moreover, apart from the starting points, the main use of the reference points in NSGA-III is to determine, after the first nondomination fronts are obtained from NSGA-II [72], the individuals to be selected from the last considered front, so that the population reaches a predefined number of individuals. This is not a problem we face because we know there is only one point (or several points, but at the same coordinates) in the Pareto front that would satisfy our constraint of full coverage at minimal cost. Hence, we do not use the Pareto front as a set of solutions, even if we intend to use individuals in the Pareto front as intermediate steps to reach relevant solutions.

Regarding MOSA [190], trade-offs obtained from approximating the Pareto front are only used for maintaining diversity during the search, which is similar to what we intend to do. But, as opposed to the use case tackled by MOSA, in our case determining inputs covering a given objective is straightforward. Indeed, for each objective, we can easily determine inputs that are able to cover it (Section 3.3.2). Hence, individuals ensuring the coverage of the objectives are easy to obtain, while the hard part of our problem is to determine a combination of inputs able to cover all the objectives at a minimal cost. Hence, even if MOSA may find a reasonable solution, because it focuses on inputs individually covering an objective and not on their collective coverage and cost, it is unlikely to find the best solution.

Hence, we propose a novel genetic algorithm, named MOCCO. We take inspiration from MOSA [190] by considering two populations: 1) a population of solutions (like MOSA's archive), called the *roofers* because they cover all the objectives (Section 3.6.3), and 2)

a population of individuals on the Pareto front (Section 3.3.4), called the *misers* because they minimize the cost, while not covering all objectives. Like NSGA-III and MOSA, MOCCO has to tackle challenges raised by many-objective problems (Section 3.2.4).

To address challenge 1, we take inspiration from the whole suite approach [87] which counts covered branches as a single objective, by defining the *exposure* as the sum of the coverage objective functions (Section 3.3.3):

$$exposure(I) \stackrel{\text{def}}{=} \sum_{bl_i \in Coverage_{obj}(C)} f_{bl_i}(I)$$

Since $f_{bl_i}(.)$ is zero when the objective bl_i is covered, the larger the exposure, the smaller the input set coverage. As described in Section 3.3.2, we do not use the exposure as objective because we want to distinguish between input blocks. But we use it as a weight when randomly selecting a parent amongst the misers (Section 3.8.3), so that the further away a miser is from complete coverage, the less likely it is to be selected. That way, we aim to benefit from the large number of dimensions to avoid getting stuck in a local optimum and to have a better convergence rate [122], while still focusing the search on the region of interest.

Since we want to deeply explore this particular region, we do not need to preserve diversity over the whole Pareto front. Therefore, we do not use diversity operators, avoiding challenge 2.

Finally, we address challenge 3 by 1) restricting the recombination operations and 2) tailoring them to our problem, as follows:

1. A crossover between roofers can only happen during the first generations, when no miser is present in the population. After the first miser is generated, crossover (Section 3.8.4) is allowed only between a roofer and a miser. Hence, the roofer parent provides full coverage while the miser parent provides almost full coverage at low cost. Moreover, because of how the objective functions are computed (Section 3.3.3), the not-yet-covered objectives are likely to be covered in an efficient way. That way, we hope to increase our chances of obtaining offspring with both large coverage and low cost.

2. Not only is our recombination strategy designed to be computationally efficient (by minimizing the number of introduced redundancies), but we exploit our knowledge of input coverage to determine a meaningful crossover between parents, with inputs from one parent for one half of the objectives and inputs from the other parent for the other half.

3.8.2 Population Initialization

During the search, because we need diversity to explore the search space, we consider a population (with size $n_{size} \geq 2$) of the least costly individuals generated so far that satisfy

full coverage. We call *roofers* such individuals, by analogy with covering a roof, and we denote $\mathit{Roofers}(n)$ the roofer population at generation n.

But focusing only on the roofers would prevent us to exploit the least expensive solutions obtained in the Pareto front while trying to minimize the connected component (Section 3.7.6). Instead, inputs that do not cover all the objectives, and will thus not be retained for the final solution, but are efficient at minimizing cost, are thus useful as intermediary steps towards finding an efficient solution. Hence, we maintain a second population, formed by individuals that are non-dominated so far and minimize cost while not covering all the objectives. We call such individuals *misers*, because they focus on cost reduction more than objective coverage, and we denote $\mathit{Misers}(n)$ the miser population at generation n.

The reason for maintaining two distinct populations is to restrict the crossover strategy (Section 3.8.4) so that (in most cases) one parent is a roofer and one parent is a miser. Since misers prioritize cost over coverage, a crossover with a miser tends to reduce cost. Because roofers prioritize coverage over cost, a crossover with a roofer tends to increase coverage. Hence, with such a strategy, we intend to converge towards a solution minimizing cost and maximizing coverage.

For both populations, we want to ensure that the individuals are reduced (Section 3.3.3), i.e., they contain no redundant inputs. Hence, during the initialization and updates of these populations, we ensure that removal steps are performed. Because, as detailed in the following, the number of redundant inputs obtained for each generation is small, the optimal order of removal steps can be exhaustively computed. We denote by $\mathit{reduce}(I)$ the input set I after these removal steps. This limits the exploration space, since non-reduced input sets are likely to have a large cost and hence to be far away for the utopia point of full coverage at no cost we intend to focus on.

Algorithm 3 Roofer population initialization.

```
1:  procedure initRoofers(C, n_size)
2:      Roofers ← ∅
3:      while |Roofers| < n_size do
4:          I ← ∅
5:          while Cover(I) ≠ Coverage_obj(C) do
6:              bl ← select(Coverage_obj(C) \ Cover(I), P_unif)
7:              in ← select(Inputs(bl), P_init)
8:              I ← I ∪ {in}
9:              I ← reduce(I)
10:         if I ∉ Roofers then
11:             Roofers ← Roofers ∪ {I}
12:     return Roofers
```

The miser population is initially empty, i.e., $\mathit{Misers}(0) \stackrel{\text{def}}{=} \varnothing$, as misers are generated during the search through mutations (Section 3.8.6). We detail in Algorithm 3 how the roofer population $\mathit{Roofers}(0)$ is initialized, where $\mathit{select}(X, \mathrm{P})$ randomly selects one element in X using distribution P, $\mathrm{P}_{\mathit{unif}}$ denotes the uniform distribution, and $\mathrm{P}_{\mathit{init}}$ denotes the

following distribution:

$$\mathrm{P}_{init}(in_1) \stackrel{\text{def}}{=} \frac{\frac{1}{1+occurrence(in_1)}}{\sum_{in_2 \in Inputs(bl)} \frac{1}{1+occurrence(in_2)}}$$

where $occurrence(in)$ denotes the number of times the input in was selected in the roofer population so far. This distribution ensures that inputs that were not selected so far are more likely to be selected, so that the initial roofer population can be more diverse. Note that computing $reduce(I)$ is tractable since adding a new input can only affect the redundancy of inputs that overlap with it (Section 3.7.5).

3.8.3 Parents Selection

For each generation n, parents are selected as follows. If $Misers(n) \neq \varnothing$, then one parent is selected from the miser population and one from the roofer population. Otherwise, two distinct parents are selected from the roofer population. A parent $I_1 \in Misers(n)$ is randomly selected from the miser population using the following distribution:

$$\mathrm{P}_{misers}(I_1) \stackrel{\text{def}}{=} \frac{\frac{1}{exposure(I_1)}}{\sum_{I_2 \in Misers(n)} \frac{1}{exposure(I_2)}}$$

where the exposure is defined in Section 3.8.1. The purpose of this distribution is to ensure that input sets with large coverage or at least large potential (Section 3.3.3) are more likely to be selected. A parent $I_1 \in Roofers(n)$ is randomly selected from the roofer population using the following distribution:

$$\mathrm{P}_{roofers}(I_1) \stackrel{\text{def}}{=} \frac{\frac{1}{cost(I_1)}}{\sum_{I_2 \in Roofers(n)} \frac{1}{cost(I_2)}}$$

The purpose of this distribution is to ensure that less costly input sets are more likely to be selected.

3.8.4 Parents Crossover

After selecting two distinct parents I_1 and I_2, we detail how they are used to generate the offspring I_3 and I_4. Our crossover strategy exploits the fact that, for each objective bl to cover, it is easy to infer inputs in $Inputs(bl)$ able to cover bl (Section 3.3.2). For each crossover, we randomly split the objectives in two halves O_1 and O_2 such that $O_1 \cup O_2 = Coverage_{obj}(C)$ and $O_1 \cap O_2 = \varnothing$. We consider here a balanced split, to prevent cases where one parent massively contributes to offspring coverage.

Then, we use this split to define the crossover: inputs in the connected component C are split between $S_1 \stackrel{\text{def}}{=} Inputs(O_1)$, the ones covering the first half of the objectives,

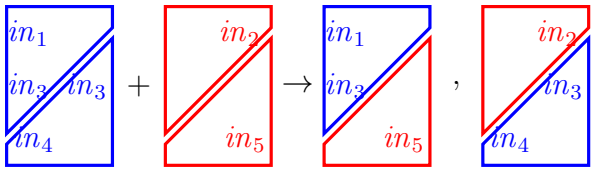

Figure 3.5: Crossover Example

and $S_2 \stackrel{\text{def}}{=} \mathit{Inputs}(O_2)$, the ones covering the second half. Note that some inputs may cover objectives both in O_1 and O_2, so we call the *edge* of the split the intersection $S_1 \cap S_2$. Because we assume both parents are reduced, this means that redundant inputs can only happen at the edge of the split. The genetic material of both parents I_1 and I_2 is then split in two parts: inputs in S_1 and inputs in S_2, as follows:

$$I_1 = (I_1 \cap S_1) \cup (I_1 \cap S_2)$$
$$I_2 = (I_2 \cap S_1) \cup (I_2 \cap S_2)$$

Then, these parts are swapped to generate the offspring, as follows:

$$I_3 \stackrel{\text{def}}{=} (I_1 \cap S_1) \cup (I_2 \cap S_2)$$
$$I_4 \stackrel{\text{def}}{=} (I_2 \cap S_1) \cup (I_1 \cap S_2)$$

For illustration purpose we consider in Figure 3.5 a small connected component $C = \{in_1, in_2, in_3, in_4, in_5\}$ and the following split for the inputs: $\mathit{Inputs}(O_1) = \{in_1, in_2, in_3\}$ and $\mathit{Inputs}(O_2) = \{in_3, in_4, in_5\}$. The edge of the split is $\mathit{Inputs}(O_1) \cap \mathit{Inputs}(O_2) = \{in_3\}$. The offspring of parents $I_1 = \{in_1, in_3, in_4\}$ and $I_2 = \{in_2, in_5\}$ is $I_3 = \{in_1, in_3, in_5\}$ and $I_4 = \{in_2, in_3, in_4\}$.

3.8.5 Offspring Mutation

Each gene of an offspring I corresponds to an input $in \in C$, the gene value being 1 if $in \in I$ and 0 otherwise. A mutation happens when this gene value is changed, hence mutation randomly adds or removes one input from an offspring.

The crossover (at the edge of the split) and mutation (when an input is added) steps may result in inputs being redundant in the offspring. Since redundancies in the connected component were already reduced by IMPRO (Section 3.7.6) and changes in redundancy could happen only amongst neighbors (for the overlapping relation) of the changed inputs (Section 3.7.5), we only expect a few redundant inputs. Therefore, removal steps can be exhaustively computed to replace each offspring I by its reduced counterpart $\mathit{reduce}(I)$ (Section 3.8.2).

3.8.6 Population Update

We detail how the offspring is used to obtain the populations $\mathit{Roofers}(n+1)$ and $\mathit{Misers}(n+1)$ at generation $n+1$.

First, we discard any offspring that is a duplicate of an individual already present in either $Roofers(n)$ or $Misers(n)$. Indeed, the duplication of individuals would only result in altering their weight for being selected as parents (thus, the intended procedure), reducing roofer diversity, and increasing the number of miser comparisons (as detailed below).

If a remaining offspring I_1 covers all the objectives in $Coverage_{obj}(C)$, then it is a candidate for the roofer population. Otherwise, it is a candidate for the miser population.

For each candidate I_1 for the roofer population, MOCCO computes its cost. If $cost(I_1) \leq \max\{cost(I) \mid I \in Roofers(n)\}$, then it selects the most costly roofer I_2 (or, in case of a tie, one of the most costly roofers), removes I_2 from the population, and adds I_1. Otherwise, I_1 is rejected. Note that we chose $\leq$ instead of $<$ for the above cost criterion because, in case of a tie, we prefer to evolve the population instead of maintaining the status quo, to increase the odds of exploring new regions of the search space.

For each candidate I_1 to the miser population, we compute its fitness vector $F_C(I_1)$ (Section 3.7.6). Then, for each $I_2 \in Misers(n)$ we compare $F_C(I_1)$ and $F_C(I_2)$. If $I_2 \succ I_1$ in the sense of Pareto-dominance (Section 3.2.4), then we stop the process and I_1 is rejected. If $I_1 \succ I_2$, then I_2 is removed from $Misers(n)$. That way, we ensure that the miser population contains only non-dominated individuals. After completing the comparisons, if the process was not stopped, then I_1 itself is non-dominated, so it is added to the miser population. In that case, $F_C(I_1)$ is stored for future comparisons.

3.8.7 Termination

MOCCO repeats the process until it reaches a fixed number of generations or exhausts a given time budget. Then, amongst the least costly roofers (several may have the same cost), it randomly selects one individual I_C as solution to our sub-problem (Section 3.7.6). I_C covers all the objectives and, amongst the input sets covering those objectives, I_C has the smallest cost encountered during the search.

3.9 Step 6: Data Post-Processing

The set I_{necess} was initially empty (Section 3.7.2) and then accumulated necessary inputs each time redundancy was determined (Section 3.7.3). After removing inputs and reducing the objectives to be covered accordingly (Section 3.7), IMPRO obtained a set I_{search} of remaining inputs and objectives. Then, IMPRO divided the remaining problem into sub-problems (Section 3.7.6), one for each connected component C. Finally, for each connected component C, the corresponding sub-problem was solved using MOCCO (Section 3.8), obtaining the corresponding minimized component I_C. At the end of the search, AIM merges inputs from each minimized component I_C with the necessary inputs I_{necess} to obtain a *minimized input set* I_{final} as solution to our initial problem (Section 3.3.4):

$$I_{final} \stackrel{\text{def}}{=} I_{necess} \cup \bigcup_{C \in Comps(I_{search})} I_C$$

3.10 Empirical Evaluation

In this section, we report our results on the assessment of our approach with two Web systems. We investigate the following Research Questions (RQs):

RQ1 What is the vulnerability detection effectiveness of AIM, compared to alternatives? This research question aims to determine if and to what extent AIM reduces the effectiveness of MST by comparing the vulnerabilities detected between the initial and the minimized input sets. Also, we further compare the vulnerability detection rate of AIM with simpler alternative approaches.

RQ2 What is the input set minimization effectiveness of AIM, compared to alternatives? This research question aims to analyze the magnitude of minimization in terms of the number of inputs, cost (Sections 3.3.1 and 3.5.1), and execution time for the considered MRs, both for AIM and alternative approaches.

RQ3 What is the input set minimization effectiveness of MOCCO, compared to alternatives? This research question aims to determine the particular contribution of the novel MOCCO genetic algorithm in minimizing cost while preserving vulnerability detection, by comparing it to alternative approaches for different time budgets.

3.10.1 Experiment Design

Subjects of the Study

To assess our approach with MRs and input sets that successfully detect real-world vulnerabilities, we rely on the same input sets and settings as *MST-wi* [47].

The targeted Web systems under test are Jenkins [77] and Joomla [2]. Jenkins is a leading open source automation server while Joomla is a content management system (CMS) that relies on the MySQL RDBMS and the Apache HTTP server. We chose these Web systems because of their plug-in architecture and Web interface with advanced features (such as Javascript-based login and AJAX interfaces), which makes Jenkins and Joomla good representatives of modern Web systems.

Further, these systems present differences in their output interface and input types that, since inputs and outputs are key drivers for our approach, contribute to improve the generalizability of our results. Concerning outputs, Joomla is a CMS where Web pages tend to contain a large amount of static text that differ in every page, while Jenkins provides mainly structured content that may continuously change (e.g., seconds from the last execution of a Jenkins task). The input interfaces of Jenkins are mainly short forms and buttons whereas the inputs interfaces of Joomla often include long text areas and several selection interfaces (e.g., for tags annotation).

The selected versions of Jenkins and Joomla—2.121.1 and 3.8.7, respectively—are affected by known vulnerabilities that can be triggered from the Web interface; we describe them in Section 3.10.1.

The input set provided in the *MST-wi*'s replication package has been collected by running Crawljax with, respectively, four users for Jenkins and six users for Joomla having different roles, e.g., admin. For each role, Crawljax has been executed for a maximum of 300 minutes, to prevent the crawler from running indefinitely, thereby avoiding excessive resource consumption. Further, to exercise features not reached by Crawljax, a few additional Selenium [214]-based test scripts (four for Jenkins and one for Joomla) have been added to the input set. In total, we have 160 initial inputs for Jenkins and 148 for Joomla, which are all associated to a unique identifier.

Since MOCCO assumes redundancy in the input set to be already reduced by IMPRO (e.g., to ensure the reduction step after mutation is tractable), we do not consider the initial input sets for RQ3 but their reduced versions. Thus, we run the double-clustering (Section 3.6) and problem reduction (Section 3.7) steps to obtain a set of necessary inputs and several connected components. Then, the input set for RQ3 is the union of these necessary inputs and connected components, called a *reduced input set*. We repeat this process several times (to reduce the impact of randomness), thus obtaining several reduced input sets.

Finally, since we know necessary inputs should be part of the solution and minimization can be performed independently on each connected component (Section 3.7), we can exhaustively search the best order of removal steps for each connected component, in order to determine the *optimal solution* for each reduced input set. Thus, we can compare the solutions obtained by different search algorithms for different time budgets to this optimal solution. Note that this approach is tractable for the considered connected components only because they are small, but it is intractable for larger input sets, e.g., the initial or reduced input sets.

Security Vulnerabilities

The replication package for *MST-wi* [57] includes 76 metamorphic relations (MRs). These MRs can identify nine vulnerabilities in Jenkins and three vulnerabilities in Joomla using the initial input set, as detailed in Tables 3.2 and 3.3, respectively.

For both tables, the first column contains, when available, the CVE identifiers of the considered vulnerabilities. The password aging problem (for both Jenkins and Joomla) and weak password (for Jenkins) are vulnerabilities that were identified during the *MST-wi* study [47] and therefore do not have CVE identifiers. The second column provides a short description of the vulnerabilities. The third column reports the CWE ID for each vulnerability. We present two CWE IDs (e.g., CWE 863 and 280) in cases where the CVE report denotes a general vulnerability type (e.g., CWE 863 for incorrect authorization [153]), though a more precise identification (e.g., CWE 280 concerning improper handling of privileges that may result in incorrect authorization) could be applied. Since

Table 3.2: Jenkins Vulnerabilities.

CVE	Description	Vulnerability Type	Input Identifiers
CVE-2018-1000406 [149]	In the file name parameter of a Job configuration, users with Job / Configure permissions can specify a relative path escaping the base directory. Such path can be used to upload a file on the Jenkins host, resulting in an arbitrary file write vulnerability.	CWE_22	160
CVE-2018-1000409 [150]	A session fixation vulnerability prevents Jenkins from invalidating the existing session and creating a new one when a user signed up for a new user account.	CWE_384	112, 113, 114
CVE-2018-1999003 [153]	Jenkins does not perform a permission check for URLs handling cancellation of queued builds, allowing users with Overall / Read permission to cancel queued builds.	CWE_280, CWE_863	116, 157
CVE-2018-1999004 [154]	Jenkins does not perform a permission check for the URL that initiates agent launches, allowing users with Overall / Read permission to initiate agent launches.	CWE_863, CWE_285	2, 116
CVE-2018-1999006 [155]	A exposure of sensitive information vulnerability allows attackers to determine the date and time when a plugin was last extracted.	CWE_200, CWE_668	33, 55, 57, 61, 62, 63, 64, 75, 107, 108, 110, 135, 136, 156, 160
CVE-2018-1999046 [156]	Users with Overall / Read permission are able to access the URL serving agent logs on the UI due to a lack of permission checks.	CWE_200	2, 116
CVE-2020-2162 [157]	Jenkins does not set Content-Security-Policy headers for files uploaded as file parameters to a build, resulting in a stored XSS vulnerability.	CWE_79	1, 18, 19, 23, 26, 75, 156, 158
Password aging problem in Jenkins	Jenkins does not integrate any mechanism for managing password aging; consequently, users aren't incentivized to update passwords periodically.	CWE_262	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 18, 19, 20, 22, 23, 24, 25, 26, 27, 28, 30, 32, 110, 33, 34, 35, 38, 39, 41, 42, 43, 44, 45, 46, 47, 58, 61, 62, 64, 65, 66, 69, 70, 71, 73, 74, 75, 104, 108, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 133, 134, 143, 145, 146, 159, 160
Weak password in Jenkins	Jenkins does not require users to have strong passwords, which makes it easier for attackers to compromise user accounts.	CWE_521	112, 113, 114

the 12 considered vulnerabilities are associated to nine different CWE IDs and each vulnerability has a unique CWE ID, we can conclude that the selected subjects cover a diverse set of vulnerability types, thus further improving the generalizability of our results.

The last column in Tables 3.2 and 3.3 lists identifiers for inputs which were able to trigger the vulnerability using one of the corresponding MRs. For instance, one can detect vulnerability CVE-2018-1999046 in Jenkins by running the MR written for CWE_200 with inputs 2 or 116. For the first two Joomla vulnerabilities, two inputs need to be present at the same time in the input set in order to trigger the vulnerability because, as opposed to most MRs, the corresponding MRs require two source inputs to generate follow-up inputs. For instance, to detect vulnerability CVE-2018-17857 in Joomla, one needs input 1 and at least one input amongst inputs 22, 23, 24, or 25.

Table 3.3: Joomla Vulnerabilities.

CVE	Description	Vulnerability Type	Input Identifiers
CVE-2018-11327 [151]	Inadequate checks allow users to see the names of tags that were either unpublished or published with restricted view permission.	CWE_200	37 with 22, 23, 24, 25, 50
CVE-2018-17857 [152]	Inadequate checks on the tag search fields can lead to an access level violation.	CWE_863	1 with 22, 23, 24, 25
Password aging problem in Joomla	Joomla does not integrate any mechanism for managing password aging; consequently, users aren't incentivized to update passwords periodically.	CWE_262	2, 3, 5, 6, 7, 8, 11, 12, 15, 17, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 66, 110, 144, 146

AIM configurations

AIM can be configured in different ways to obtain a minimized input set from an initial input set. Such a *configuration* consists in a choice of distance function and algorithm for output clustering (Section 3.6.2), and a choice of algorithm for action clustering (Section 3.6.3).

For the sake of conciseness, in Tables 3.5 to 3.11, we denote each configuration by three letters, where L and B respectively denote the Levenshtein and Bag distances, and K, D, and H respectively denote the K-means, DBSCAN, and HDBSCAN clustering algorithms. For instance, BDH denotes that Bag distance and DBSCAN were used for output clustering, and then HDBSCAN for action clustering. This notation are summarized in Table 3.4.

AIM performs Silhouette analysis (Section 3.6.1) to determine the hyper-parameters required for these clustering algorithms. We considered the same ranges of values for the hyper-parameters in both output clustering (Section 3.6.2) and action clustering steps (Section 3.6.3). For K-means, we select the range $[1, 70]$ for the number of clusters k. In the case of DBSCAN, the range for the distance threshold ϵ is $[2, 10]$ for Jenkins and $[3, 15]$ for Joomla. The range is larger for Joomla because Joomla has a larger number of Web pages than Jenkins. Finally, the range for the minimum number of neighbors n is $[1, 5]$ for both systems. For HDBSCAN, the range for the minimum number n of individuals required to form a cluster is $[2, 8]$ for both systems.

We also determine the hyper-parameters for the genetic search (Section 3.8). Related work on whole test suite generation successfully relied on a population of 80 individuals [87]. Since we reduced the problem (Section 3.7) before applying MOCCO independently to each connected component, which includes fewer inputs than the whole test suite generation [87], we experimented with a lower population size of 20 individuals. Additionally, for RQ1 and RQ2, we set the number of generations for the genetic algorithm to 100, similar to the value considered in previous work [87]. However, for RQ3, to compare the minimized input sets obtained by search algorithms with different time budgets, we use a maximum time budget of 600 seconds as the termination criterion for MOCCO, consistent with MOSA's study [190], while recording the intermediate results over time.

Table 3.4: AIM configurations and baselines for RQ1 and RQ2.

		Output clustering		Action clustering
		Distance	Algorithm	Algorithm
AIM configurations	LKK	Levenshtein	K-means	K-means
	LKD	Levenshtein	K-means	DBSCAN
	LKH	Levenshtein	K-means	HDBSCAN
	LDK	Levenshtein	DBSCAN	K-means
	LDD	Levenshtein	DBSCAN	DBSCAN
	LDH	Levenshtein	DBSCAN	HDBSCAN
	LHK	Levenshtein	HDBSCAN	K-means
	LHD	Levenshtein	HDBSCAN	DBSCAN
	LHH	Levenshtein	HDBSCAN	HDBSCAN
	BKK	Bag	K-means	K-means
	BKD	Bag	K-means	DBSCAN
	BKH	Bag	K-means	HDBSCAN
	BDK	Bag	DBSCAN	K-means
	BDD	Bag	DBSCAN	DBSCAN
	BDH	Bag	DBSCAN	HDBSCAN
	BHK	Bag	HDBSCAN	K-means
	BHD	Bag	HDBSCAN	DBSCAN
	BHH	Bag	HDBSCAN	HDBSCAN
Baselines	R	×		Random
	AK	×		K-means
	AD	×		DBSCAN
	AH	×		HDBSCAN

Baselines

For RQ1 and RQ2, we identify the following baselines against which to compare AIM configurations.

A 2016 survey reported that 57% of metamorphic testing (MT) work used Random Testing (RT) to generate source inputs [210] and, in 2021, 84% of the publications related to MT adopted traditional or improved RT methods to generate source inputs [97]. In the context of test suite minimization, random search is a straightforward baseline against which to compare AIM that is commonly used [236,252]. This baseline consists in randomly selecting a given number of inputs from the initial input set. This number is determined based on AIM runs. Each AIM run is performed using 18 different configurations (Section 3.10.1), each leading to a different minimized input set. So, for a fair comparison, we configure random search to select n inputs from the initial input, where n is the size of the largest input set produced by the 18 AIM configurations. We repeat this process, for each AIM run, to obtain the same number of input sets for random search as for AIM.

Moreover, Adaptive Random Testing (ART) was proposed to enhance the performance of RT. It is based on the intuition that inputs close to each other are more likely to have similar failure behaviors than inputs further away from each other. Thus, ART generates inputs widely spread across the input domain, in order to find failure with fewer inputs than RT [45]. ART is also commonly used in the context of test suite minimization [69,95]. It is similar to our action clustering step (Section 3.6.3), since it is based on partitioning the input space and generating new inputs in blocks that are not already covered [97].

To perform ART, we need to group inputs based on the similarity between their actions. So, we use AIM to perform action clustering directly on the initial input set instead of output classes. Then, for each cluster, we randomly select one input that covers it. Finally, we group the selected inputs to obtain an input set for the ART baseline. This algorithm

will stop when we have selected one input from each cluster, and thus, it is not limited by a time budget. Again, we repeat this process for each AIM run. Since we considered the K-means, DBSCAN, and HDBSCAN clustering algorithms, there are three variants of this baseline.

In Tables 3.5 to 3.11, R denotes the random search baseline while AK, AD, and AH denote the ART baselines, using respectively the K-means, DBSCAN, and HDBSCAN clustering algorithms. These notations are summarized in Table 3.4.

RQ3 aims at determining the contribution of the MOCCO genetic algorithm to input set minimization and comparing it with alternative approaches. We consider random search, a greedy algorithm for the set cover problem (Section 3.8), as well as MOSA and NSGA-III (Section 3.2.4), which are well-known many-objective search algorithms [29, 90, 109, 177].

We use random search to provide insights on how difficult the problem is and to help assess if search algorithms are necessary to solve it. Random search starts with an empty input set and, for each iteration, randomly selects one input from the reduced input set (Section 3.10.1). This process continues until all the objectives are covered.

The greedy algorithm for the set cover problem [233] is one of the best-possible polynomial time approximation algorithm for this problem, with a tight bound on the cost of the solution for the unweighted variant of the problem [216], matching theoretical bounds [76], and that can be efficiently adapted to the weighted variant [233]. Since the weighted set cover problem is close to our problem (Section 3.8), we adapt this greedy algorithm to obtain a minimized input set from the reduced input set. The minimized input set is initially empty. For each iteration, the input with the best cost effectiveness is selected, where the cost effectiveness of input *in* is computed as $\frac{|Cover(in) \cap Uncovered|}{cost(in)}$. Recall that $Cover(in)$ and $cost(in)$ are defined in Section 3.3.1, and *Uncovered* denotes the set of the objectives not yet covered by selected inputs. This process continues until all the objectives are covered.

The initial population for both MOSA and NSGA-III is the reduced input set. For the other parameters, such as population size, mutation rate, and (for NSGA-III) the number of reference points (Section 3.2.4), we use the default values recommended in the original studies [73, 98, 190]. More precisely, the population size for MOSA is set to 50, while it is automatically computed for NSGA-III based on the number of reference points. We set the crossover rate to 1 for both MOSA and NSGA-III, the same as MOCCO. Finally, we use the same objective functions and mutation operator for both MOSA and NSGA-III, as in MOCCO. Finally, for MOCCO and all baselines, we use the same termination criterion as in MOSA's study [190], allocating a maximal time budget of 600 seconds for all search algorithms, while recording intermediate results to determine the minimized input sets for smaller time budgets.

Evaluation Metrics

To reduce the impact of randomness in our experiment, each configuration and baseline was run 50 times on each system, obtaining one minimized input set for each run. Moreover, for the sake of performance analysis, we also recorded the execution time required by AIM

to generate minimized input sets. The purpose of the metrics we use for RQ1 and RQ2 is to determine the "best" configuration to run AIM on a given system. But one cannot know, before experimenting with the target system, which configuration would be the "best" for this system. Based on the results from Jenkins and Joomla (see Section 3.10.1), we determine the overall "best" configuration, to be recommended as default for a new system.

For RQ1, we consider the vulnerabilities described in Table 3.2 for Jenkins and in Table 3.3 for Joomla. We manually investigated the results of the initial input sets to identify the inputs capable of detecting vulnerabilities in the systems under test, so that we can map inputs to vulnerabilities. For each system, we consider that a vulnerability is detected by a minimized input set if it contains at least one input or pair of inputs able to trigger this vulnerability. For the first two Joomla vulnerabilities requiring pairs of inputs, the vulnerability is detected if both inputs are present in the input set. Hence, for each configuration or baseline, our metric is the *vulnerability detection rate (VDR)*, i.e., the total number of vulnerabilities detected by the minimized input sets obtained for the 50 runs, divided by the total number of vulnerabilities detected by the corresponding initial input sets. If VDR is 100%, then we say the configuration or baseline leads to full vulnerability coverage. The overall "best" configuration for Jenkins and Joomla, regarding vulnerability detection, should have a large VDR for both systems, ideally 100%. For each system, we reject configurations and baselines which do not lead to full vulnerability coverage, and then compare the remaining ones to answer RQ2.

RQ2 aims at evaluating the effectiveness of AIM in minimizing the initial input set. Our goal is to identify the AIM configuration generating minimized input sets leading to the minimal execution time for the 76 considered MRs, across the two case studies, and reporting on the execution time saved, compared to executing *MST-wi* on the full input set. But, to have a fair comparison between MRs execution time obtained respectively with the initial and minimized input sets, we have to take into account the AIM execution time required to minimize the initial input set. Thus, the input set minimization effectiveness is quantified as the sum of AIM execution time to obtain the minimized input set plus MRs execution time with the minimized input set, divided by that of the initial input set. However, since MR execution time is usually large, we cannot collect the time required to execute our 76 MRs on all the input sets generated by all AIM configurations. We estimate it would take thousands of hours for the 1800 runs, resulting from 18 configurations $\times$ 50 repetitions $\times$ 2 case study subjects. For this reason, we rely on three additional metrics, that can be inferred without executing MRs, to identify the "best" configuration. Then, we report on the input set minimization effectiveness obtained by such configuration. Further, to keep the experiment within feasible computation resources, amongst the 50 minimized input sets of the best configuration, we select one with a median cost (Sections 3.3.1 and 3.5.1), to be representative of the 50 runs.

To determine the "best" AIM configuration, we consider the size of the generated input set (i.e., the number of inputs in it), its cost (Sections 3.3.1 and 3.5.1), and the time required by the configuration to generate results. Input set size is a direct measure of effectiveness, while cost is an indirect measure, specific to our approach, that is linearly correlated with MR execution time (Section 3.2.1). For these three metrics (size, cost,

AIM execution time), we compare, for each system, the AIM configurations and baselines leading to full vulnerability coverage. More precisely, for each metric, we denote by M_i the value of the metric obtained for the i^{th} approach (AIM configurations or baseline); the 50 runs of approach i leading to a sample containing 50 data points.

To compare two samples for M_1 and M_2, we perform a Mann-Whitney-Wilcoxon test, which is recommended to assess differences in stochastic order for software engineering data analysis [36]. This is a non-parametric test of the null hypothesis that $\mathrm{P}(M_1 > M_2) = \mathrm{P}(M_1 < M_2)$, i.e., M_1 and M_2 are stochastically equal [232]. Hence, from M_1 and M_2 samples, we obtain the p-value p indicating how likely is the observation of these samples, assuming that M_1 and M_2 are stochastically equal. If $p \leq 0.05$, we consider it is unlikely that M_1 and M_2 are stochastically equal.

To assess practical significance, we also consider a metric for effect size. An equivalent reformulation of the null hypothesis is $\mathrm{P}(M_1 > M_2) + 0.5 \times \mathrm{P}(M_1 = M_2) = 0.5$, which can be estimated by counting in the samples the number of times a value for M_1 is larger than a value for M_2 (ties counting for 0.5), then by dividing by the number of comparisons. That way, we obtain the Vargha and Delaney's A_{12} metric [232] which, for the sake of conciseness, we simply denote A in Tables 3.6 to 3.11. A is considered to be a robust metric for representing effect size in the context of non-parametric methods [110]. A ranges from 0 to 1, where $A = 0$ indicates that $\mathrm{P}(M_1 < M_2) = 1$, $A = 0.5$ indicates that $\mathrm{P}(M_1 > M_2) = \mathrm{P}(M_1 < M_2)$, and $A = 1$ indicates that $\mathrm{P}(M_1 > M_2) = 1$.

RQ3 aims at determining the contribution of MOCCO to input set minimization and comparing it with alternative approaches. We use the results of the double-clustering and problem reductions steps to obtain reduced input sets (Section 3.10.1) from the 50 runs of the "best" configuration. For each reduced input set, each considered algorithm (MOCCO or a baseline) is executed to obtain the corresponding minimized input set. Then, the minimized input sets are checked to determine if the algorithms lead to full vulnerability coverage for this run, and their cost (Sections 3.3.1 and 3.5.1) is recorded. Again, we denote by M_i the cost for the i^{th} approach. The 50 runs of approach i yield a sample containing 50 data points. As opposed to RQ2, where any run from a configuration/baseline can be compared to any run of another configuration/baseline, we want to compare the cost of the n^{th} minimized input set for an algorithm to the cost of the n^{th} minimized input set of another algorithm, since they are both obtained from the same n^{th} reduced input set. Hence, to compare two samples for M_1 and M_2, we perform a Wilcoxon signed-rank test, which is a non-parametric paired test [36], with a level of significance of 0.05.

To assess practical significance, we also consider a metric for effect size. Metrics for this test are often defined in terms of the positive-rank sum R^+ and the negative-rank sum R^- [74, 107]. Similarly to the Vargha and Delaney's A_{12} metric [232] we used for the Mann-Whitney-Wilcoxon test to answer RQ2, we use for RQ3 the effect size $E \stackrel{\text{def}}{=} \frac{R^+}{R^+ + R^-}$, so that E ranges from 0 to 1, where $E = 0$ indicates that $M_1 < M_2$ in every case and $E = 1$ indicates that $M_1 > M_2$ in every case.

3.10.2 Empirical Results

We first describe the system configurations used to obtain our results (Section 3.10.2). To answer RQ1, we report the VDR associated with the obtained minimized input sets (Section 3.10.2). Then, we describe the effectiveness of the input set reduction of the whole AIM approach to answer RQ2 (Section 3.10.2) and of the MOCCO component to answer RQ3 (Section 3.10.2).

System Configurations

We performed all the experiments on a system with the following configurations: a virtual machine installed on professional desktop PCs (Dell G7 7500, RAM 16Gb, Intel(R) Core(TM) i9-10885H CPU @ 2.40GHz) and terminal access to a shared remote server with Intel(R) Xeon(R) Gold 6234 CPU (3.30GHz) and 8 CPU cores.

RQ1 - Detected Vulnerabilities

Results are presented in Table 3.5. Configurations and baselines that lead to full vulnerability coverage for both systems are in green, in yellow if they lead to full vulnerability coverage for one system, and in red if they never lead to full vulnerability coverage. In this chapter, we consider only the vulnerabilities in both Jenkins and Joomla that were detected by *MST-wi* (see Table 2.23). As shown in Table 3.2 and Table 3.3, there are 9 vulnerabilities in Jenkins and 3 vulnerabilities in Joomla that were previously detected by *MST-wi*. Each AIM configuration is executed 50 times to reduce the effect of randomness in our experiments. We consider an AIM configuration to achieve full vulnerability coverage on Jenkins if it achieves $9 * 50 = 450$ vulnerability detections across all runs. Similarly, full vulnerability coverage on Joomla across all runs is reached if the configuration achieves $3 * 50 = 150$ vulnerability detections. The execution time of the AIM configurations ranges from 15 to 24 minutes on Jenkins and from 25 to 47 minutes on Joomla. We conclude that such variation across configurations is not significant compared to the time required to execute MRs.

First, note that **the choice of distance function for output clustering does not have a significant impact on vulnerability coverage**. Indeed, apart from LDH and BDH, the results using the Levenshtein or Bag distances are fairly similar (e.g., both LKK and BKK discover 450 vulnerabilities in Jenkins) and seem to only depend on the choice of clustering algorithms. This indicates that the order of words in a Web page is not a relevant distinction when performing clustering for vulnerability coverage. Considering now LDH and BDH, taking into account the order of words can even be detrimental, since they perform equally poorly for Joomla but they differ for Jenkins, where only BDH leads to full vulnerability coverage.

Second, **the choice of clustering algorithm for action clustering seems to be the main factor determining vulnerability coverage**. Configurations using DBSCAN as algorithm for the action clustering step never lead to full vulnerability coverage for any

Table 3.5: Coverage of the Jenkins and Joomla vulnerabilities after 50 runs of each configuration and baseline.

Vulnerability Coverage	System Under Test			
	Jenkins		Joomla	
Configurations or baselines	Nb of detected vulnerabilities	VDR	Nb of detected vulnerabilities	VDR
LKK	450	100.0%	146	97.3%
LKD	371	82.4%	50	33.3%
LKH	379	84.2%	150	100.0%
LDK	450	100.0%	150	100.0%
LDD	400	88.9%	50	33.3%
LDH	400	88.9%	50	33.3%
LHK	450	100.0%	100	66.7%
LHD	403	89.6%	100	66.7%
LHH	447	99.3%	100	66.7%
BKK	450	100.0%	133	88.7%
BKD	403	89.6%	50	33.3%
BKH	410	91.1%	150	100.0%
BDK	450	100.0%	150	100.0%
BDD	338	75.1%	50	33.3%
BDH	450	100.0%	50	33.3%
BHK	450	100.0%	100	66.7%
BHD	404	89.8%	100	66.7%
BHH	428	95.1%	100	66.7%
R	339	75.3%	74	49.3%
AK	447	99.3%	125	83.3%
AD	77	17.1%	22	14.7%
AH	350	77.8%	68	45.3%

system. This indicates that this clustering algorithm poorly fits the data in the input space. This is confirmed by the results obtained for the AD baseline, which only uses DBSCAN on the input space and performs the worst (amongst baselines and AIM configurations) regarding vulnerability coverage. After investigation, the minimized input sets acquired for AD are much smaller compared to those obtained for the other baseline methods. These results cannot be explained by the hyper-parameter as we employed a large range of values (Section 3.10.1). We conjecture that DBSCAN merges together many action clusters even when the URLs involved in these actions are distinct.

On the other hand, **configurations using K-means for the action clustering step always lead to full vulnerability coverage for Jenkins and lead to the largest vulnerability coverage for Joomla**. This is confirmed by the results obtained for the AK baseline, which only uses K-means on the input space and performs the best (amongst baselines) regarding vulnerability coverage. Indeed, even if this configuration does not lead to full vulnerability coverage, it is very close. In fact, even if it tends to perform worse than AIM configurations that use K-means for action clustering, it tends to perform better than AIM configurations that do not use K-means for action clustering. The success of K-means in achieving better vulnerability coverage on these datasets can be attributed to its ability to handle well-separated clusters. In our case, these clusters are well-separated because of the distinct URLs occurring in the datasets.

Finally, **no baseline reached full vulnerability coverage**. On top of the already mentioned AK and AD baselines, AH performed similarly to random testing (R), indicating that the effect of the HDBSCAN algorithm for action clustering is neutral. The only AIM configuration that performed worse than random testing is BDD, combining DBSCAN

Table 3.6: Comparison of Jenkins input set sizes for configurations with full vulnerability coverage.

sizes		LKK	**LDK**	LHK	BKK	**BDK**	BDH	BHK
LKK	p		5.4×10^{-15}	5.1×10^{-1}	4.2×10^{-1}	8.8×10^{-1}	3.1×10^{-20}	7.2×10^{-1}
	A		0.05	0.54	0.55	0.49	1.0	0.48
LDK	p	5.4×10^{-15}		1.8×10^{-17}	1.4×10^{-15}	1.8×10^{-14}	3.2×10^{-20}	3.8×10^{-14}
	A	0.95		0.99	0.96	0.94	1.0	0.94
LHK	p	5.1×10^{-1}	1.8×10^{-17}		9.8×10^{-1}	3.4×10^{-1}	3.0×10^{-20}	1.6×10^{-1}
	A	0.46	0.01		0.5	0.44	1.0	0.42
BKK	p	4.2×10^{-1}	1.4×10^{-15}	9.8×10^{-1}		4.1×10^{-1}	3.2×10^{-20}	2.1×10^{-1}
	A	0.45	0.04	0.5		0.45	1.0	0.43
BDK	p	8.8×10^{-1}	1.8×10^{-14}	3.4×10^{-1}	4.1×10^{-1}		3.2×10^{-20}	6.9×10^{-1}
	A	0.51	0.06	0.56	0.55		1.0	0.48
BDH	p	3.1×10^{-20}	3.2×10^{-20}	3.0×10^{-20}	3.2×10^{-20}	3.2×10^{-20}		3.2×10^{-20}
	A	0.0	0.0	0.0	0.0	0.0		0.0
BHK	p	7.2×10^{-1}	3.8×10^{-14}	1.6×10^{-1}	2.1×10^{-1}	6.9×10^{-1}	3.2×10^{-20}	
	A	0.52	0.06	0.58	0.57	0.52	1.0	

(as mentioned before, the worst clustering algorithm regarding vulnerability detection) for both output and action clustering with Bag distance. Only LDK and BDK lead to full vulnerability coverage for both Jenkins and Joomla, and hence are our candidate "best" configurations in terms of VDR. The combination of DBSCAN and K-means was very effective on our dataset since DBSCAN was able to identify dense regions of outputs and K-means allowed for further refinement, forming well-defined action clusters based on URLs.

RQ2 - Input Set Reduction Effectiveness

To answer RQ2 on the effectiveness of minimization, we compare the input set reduction of baselines and configurations for both Jenkins and Joomla. Amongst them, only the LKK, LDK, LHK, BKK, BDK, BDH, and BHK configurations lead to full vulnerability coverage for Jenkins. Their input set sizes are compared in Table 3.6, their costs in Table 3.7, and their AIM execution time in Table 3.8. Similarly, only the LKH, LDK, BKH, and BDK configurations lead to full vulnerability coverage for Joomla. Their input set sizes are compared in Table 3.9, their costs in Table 3.10, and their AIM execution time in Table 3.11. Configurations with full vulnerability coverage for both Jenkins and Joomla (i.e., LDK and BDK) are in bold.

In these six tables, configurations in each row are compared with configurations in each column. p denotes the statistical significance and A the effect size (Section 3.10.1). When $p > 0.05$, we consider the metric values obtained from the two configurations not to be significantly different, and hence the cell is left white. Otherwise, the cell is colored, either in green or red. Since we consider input set size and cost and AIM execution time, the smaller the values the better. Thus, green (resp. red) indicates that the configuration in the row is better (resp. worse) than the configuration in the column. The intensity of the color is proportional to the effect size. More precisely, the intensity is $|\delta|$, where $\delta = 2 \times A - 1$ is Cliff's delta [110]. $|\delta|$ is a number between 0 and 1, where 0 indicates the smallest intensity (the lightest color) and 1 indicates the largest intensity (the darkest color).

Table 3.7: Comparison of Jenkins input set costs for configurations with full vulnerability coverage.

costs		LKK	**LDK**	LHK	BKK	**BDK**	BDH	BHK
LKK	p		2.5×10^{-16}	5.9×10^{-5}	1.5×10^{-2}	4.4×10^{-3}	4.1×10^{-18}	5.4×10^{-3}
	A		0.02	0.73	0.64	0.67	1.0	0.66
LDK	p	2.5×10^{-16}		7.0×10^{-18}	9.5×10^{-18}	7.0×10^{-18}	4.1×10^{-18}	7.0×10^{-18}
	A	0.98		1.0	1.0	1.0	1.0	1.0
LHK	p	5.9×10^{-5}	7.0×10^{-18}		1.0×10^{-1}	2.4×10^{-1}	4.1×10^{-18}	1.4×10^{-1}
	A	0.27	0.0		0.41	0.43	1.0	0.41
BKK	p	1.5×10^{-2}	9.5×10^{-18}	1.0×10^{-1}		5.8×10^{-1}	4.1×10^{-18}	6.1×10^{-1}
	A	0.36	0.0	0.59		0.53	1.0	0.53
BDK	p	4.4×10^{-3}	7.0×10^{-18}	2.4×10^{-1}	5.8×10^{-1}		4.1×10^{-18}	8.2×10^{-1}
	A	0.33	0.0	0.57	0.47		1.0	0.49
BDH	p	4.1×10^{-18}	4.1×10^{-18}	4.1×10^{-18}	4.1×10^{-18}	4.1×10^{-18}		4.1×10^{-18}
	A	0.0	0.0	0.0	0.0	0.0		0.0
BHK	p	5.4×10^{-3}	7.0×10^{-18}	1.4×10^{-1}	6.1×10^{-1}	8.2×10^{-1}	4.1×10^{-18}	
	A	0.34	0.0	0.59	0.47	0.51	1.0	

Table 3.8: Comparison of Jenkins AIM execution times for configurations with full vulnerability coverage.

times		LKK	**LDK**	LHK	BKK	**BDK**	BDH	BHK
LKK	p		1.5×10^{-6}	1.0×10^{-11}	2.0×10^{-2}	3.8×10^{-5}	3.1×10^{-18}	1.3×10^{-9}
	A		0.78	0.89	0.63	0.74	1.0	0.85
LDK	p	1.5×10^{-6}		1.2×10^{-5}	3.9×10^{-3}	5.4×10^{-1}	2.6×10^{-18}	1.1×10^{-3}
	A	0.22		0.75	0.33	0.54	1.0	0.69
LHK	p	1.0×10^{-11}	1.2×10^{-5}		1.7×10^{-9}	1.4×10^{-3}	3.2×10^{-17}	8.7×10^{-1}
	A	0.11	0.25		0.15	0.32	0.98	0.49
BKK	p	2.0×10^{-2}	3.9×10^{-3}	1.7×10^{-9}		8.0×10^{-3}	3.0×10^{-18}	6.7×10^{-7}
	A	0.37	0.67	0.85		0.65	1.0	0.79
BDK	p	3.8×10^{-5}	5.4×10^{-1}	1.4×10^{-3}	8.0×10^{-3}		1.5×10^{-17}	1.1×10^{-2}
	A	0.26	0.46	0.68	0.35		0.99	0.65
BDH	p	3.1×10^{-18}	2.6×10^{-18}	3.2×10^{-17}	3.0×10^{-18}	1.5×10^{-17}		6.3×10^{-16}
	A	0.0	0.0	0.02	0.0	0.01		0.04
BHK	p	1.3×10^{-9}	1.1×10^{-3}	8.7×10^{-1}	6.7×10^{-7}	1.1×10^{-2}	6.3×10^{-16}	
	A	0.15	0.31	0.51	0.21	0.35	0.96	

For Jenkins, among the candidate best configurations (i.e., LDK and BDK), **BDK performed significantly better than LDK for input set size and cost**, and even if the difference is smaller for AIM execution time, the effect size is also in favor of BDK. As for the other configurations, Table 3.6 on input set sizes and Table 3.7 on input set costs consistently indicate that BDH is the best configuration while LDK is the worst configuration. The other configurations seem equivalent in terms of size. Regarding cost, LKK tends to be the second to last configuration, the other configurations being equivalent. Regarding AIM execution time in Table 3.8, the results are more nuanced, BDH is again the best configuration, but this time LKK is the worst configuration instead of LDK. BDH is the only configuration that reached full vulnerability coverage for Jenkins without using the K-means clustering algorithm and it performs significantly better than the other configurations, especially the ones involving K-means for both output and action clustering steps. This indicates, without surprise, that the K-means algorithm takes more resources to be executed. BDH did not lead to full vulnerability coverage for Joomla, so we do not consider it as a candidate for "best" configuration.

For Joomla, **BDK performed significantly better than LDK** for the considered

Table 3.9: Comparison of Joomla input set sizes for configurations with full vulnerability coverage.

sizes		LKH	**LDK**	BKH	**BDK**
LKH	p A		4.8×10^{-15} 0.06	1.3×10^{-1} 0.42	2.9×10^{-16} 0.06
LDK	p A	4.8×10^{-15} 0.94		1.1×10^{-14} 0.94	4.5×10^{-16} 0.94
BKH	p A	1.3×10^{-1} 0.58	1.1×10^{-14} 0.06		7.4×10^{-16} 0.06
BDK	p A	2.9×10^{-16} 0.94	4.5×10^{-16} 0.06	7.4×10^{-16} 0.94	

Table 3.10: Comparison of Joomla input set costs for configurations with full vulnerability coverage.

costs		LKH	**LDK**	BKH	**BDK**
LKH	p A		1.3×10^{-14} 0.06	6.9×10^{-1} 0.48	3.5×10^{-15} 0.05
LDK	p A	1.3×10^{-14} 0.94		1.3×10^{-14} 0.94	7.0×10^{-15} 0.94
BKH	p A	6.9×10^{-1} 0.52	1.3×10^{-14} 0.06		3.6×10^{-15} 0.05
BDK	p A	3.5×10^{-15} 0.95	7.0×10^{-15} 0.06	3.6×10^{-15} 0.95	

metrics. As for the other configurations, Table 3.9 for input set sizes and Table 3.10 for input set costs provide identical results, indicating that LKH and BKH dominate the others while being equivalent. Moreover, BDK dominates LDK, which is the worst configuration. The results are almost identical for AIM execution time in Table 3.11, with the small difference that LKH performs slightly better than BKH. However, LKH and BKH did not lead to full vulnerability coverage for Jenkins, as opposed to BDK and LDK.

Since we obtained similar results for both Jenkins and Joomla, **we consider BDK to be the "best" AIM configuration**. This is not surprising since Bag distance is less costly to compute than Levenshtein distance (Section 3.6.2) and we already observed that the order of words in a Web page does not appear to be a relevant distinction for vulnerability coverage (Section 3.10.2).

As mentioned in Section 3.10.2, no baseline leads to full vulnerability coverage. AD fared poorly and AH performed similarly to random testing R, but AK was much better, with 99.3% VDR for Jenkins and 83.3% for Joomla. But even if AK had reached full vulnerability coverage for both systems, it would be at a disadvantage compared to AIM configurations. Indeed, over 50 Jenkins runs, the average input set size for AK was 94.92 inputs, while it ranges from 38 inputs (40% of AK) for BDH to 74.8 inputs (79%) for LDK. The average input set cost for AK was 193 698.94 actions, while it ranges from 70 500.76 actions (36%) for BDH to 152 373.54 actions (79%) for LDK. Over 50 Joomla runs, the average input set size for AK was 70 inputs, while it ranges from 36.02 inputs (51%) for LKH to 41.46 inputs (59%) for LDK. The average input set cost for AK was 2 312 784.58 actions, while it ranges from 580 705.24 actions (25%) for LKH to 872 352.72 actions (38%) for LDK. In short, **all AIM configurations with full vulnerability coverage outperformed the best baseline AK**, which highlights the relevance of our approach in reducing the cost of testing.

Table 3.11: Comparison of Joomla AIM execution times for configurations with full vulnerability coverage.

times		LKH	**LDK**	BKH	**BDK**
LKH	p A		2.5×10^{-18} 0.0	3.6×10^{-3} 0.33	1.2×10^{-18} 0.0
LDK	p A	2.5×10^{-18} 1.0		2.8×10^{-18} 1.0	1.5×10^{-18} 1.0
BKH	p A	3.6×10^{-3} 0.67	2.8×10^{-18} 0.0		1.3×10^{-18} 0.0
BDK	p A	1.2×10^{-18} 1.0	1.5×10^{-18} 0.0	1.3×10^{-18} 1.0	

Table 3.12: Comparison of MRs execution time before and after input set minimization.

Execution time (minutes)	**Jenkins**	**Joomla**
MRs with initial input set	38 307	20 703
MRs with minimized input set	6119	3675
+ AIM execution time	22	22
= Total execution time	6141	3697
Percentage of Reduction	**84%**	**82%**

Finally, in Table 3.12, we present the results of executing the MRs using both the initial input set and the minimized input set derived from the best configuration (BDK). In total, by applying AIM, we reduced the execution time of all 76 MRs from 38 307 minutes to 6119 minutes for Jenkins and from 20 703 minutes to 3675 minutes for Joomla, using the minimized input set with median cost. Moreover, executing AIM to obtain this minimized input set required 22 minutes for both systems. Hence, we have a total execution time of 6141 minutes for Jenkins and 3697 minutes for Joomla. As a result, the ratio of the total execution time for the minimized input sets divided by the execution time for the initial input sets is 16.03% for Jenkins and 17.85% for Joomla. In other words, **AIM reduced the execution time by about 84% for Jenkins and more than 82% for Joomla**. This large reduction in execution time demonstrates the effectiveness of our approach in reducing the cost of metamorphic security testing.

RQ3 - Comparison of Search Algorithms

To answer RQ3, we consider the 50 reduced input sets obtained from the best AIM configuration, namely BDK, and we compare the cost of the minimized input sets obtained by MOCCO and baselines. The cost of the 50 minimized input sets obtained for each genetic algorithm is represented using box plots in Figures 3.6 and 3.7 for Jenkins and Joomla, respectively. The results are presented for different time budgets, ranging from 0.2 to 600 seconds, by which time greedy and Many-Objective Coverage and Cost Optimizer have converged. For both Jenkins and Joomla, **random search, greedy algorithm, and MOSA quickly converge. Random search and MOSA converge toward sub-optimal solutions**, which is expected since random search is unlikely to determine the best order of removal steps by chance, and MOSA minimizes the cost for each individual objective instead of considering the collective coverage of the selected inputs. **The greedy algorithm finds a good approximation for all runs on both systems**. It finds the

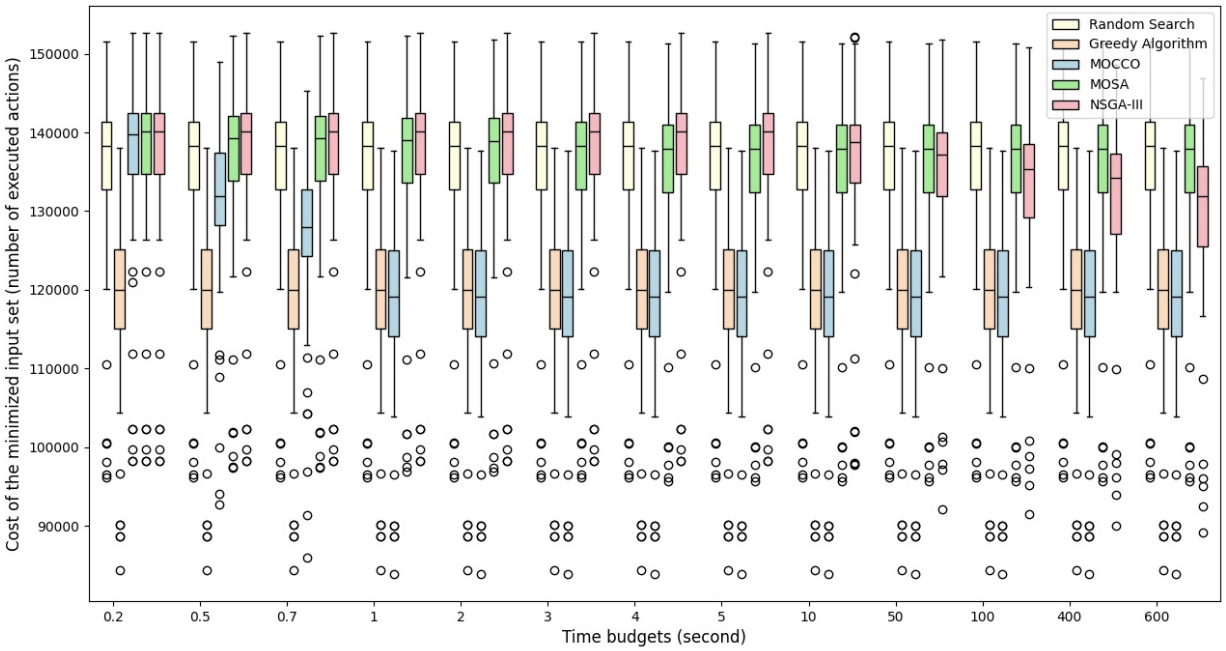

Figure 3.6: Jenkins: Cost of the minimized input sets using Random Search, Greedy Algorithm, MOCCO, MOSA, and NSGA-III under different time budgets for 50 runs of BDK.

optimal solution for most runs (41 out of 50 runs) on Joomla but for only a few runs (6 out of 50 runs) on Jenkins, likely because Jenkins reduced input sets are larger than those of Joomla. **MOCCO finds the optimal solution** (Section 3.10.1) **for all 50 runs in 1 second for Jenkins and 0.7 seconds for Joomla**, which is expected since MOCCO is designed to solve this many-objective problem (Section 3.8.1). NSGA-III slowly converges for both systems. **NSGA-III does not find the optimal solution within the 600-second time budget for Jenkins, but does so in 600 seconds for Joomla**. This is also expected since NSGA-III explores the entire Pareto front while MOCCO focuses on the region of interest, i.e, around the utopia point of full coverage at no cost. NSGA-III is the only baseline that finds the optimal solution for all runs and within the time budget, but only for one system, **while MOCCO consistently finds the optimal solution in nearly three orders of magnitude faster**.

Furthermore, we conducted statistical tests reported in Table 3.13 for Jenkins and Tables 3.14 and 3.15 for Joomla. Since NSGA-III achieves the same result as MOCCO within 600 seconds on Joomla, we also report their results at 400 seconds for a more comprehensive comparison. In these three tables, algorithms in each row are compared with algorithms in each column. p denotes the statistical significance and E the effect size (Section 3.10.1). As for RQ2, when $p > 0.05$, we consider the costs obtained from the two algorithms not to be significantly different, and hence the cell is left white. Otherwise, the cell is colored, either in green or red. Since we consider input set cost, the smaller the values the better. Thus, green (resp. red) indicates that the algorithm in the row is better (resp. worse) than the algorithm in the column. Finally, as for RQ2, the intensity of the color is computed with $|2 \times E - 1|$.

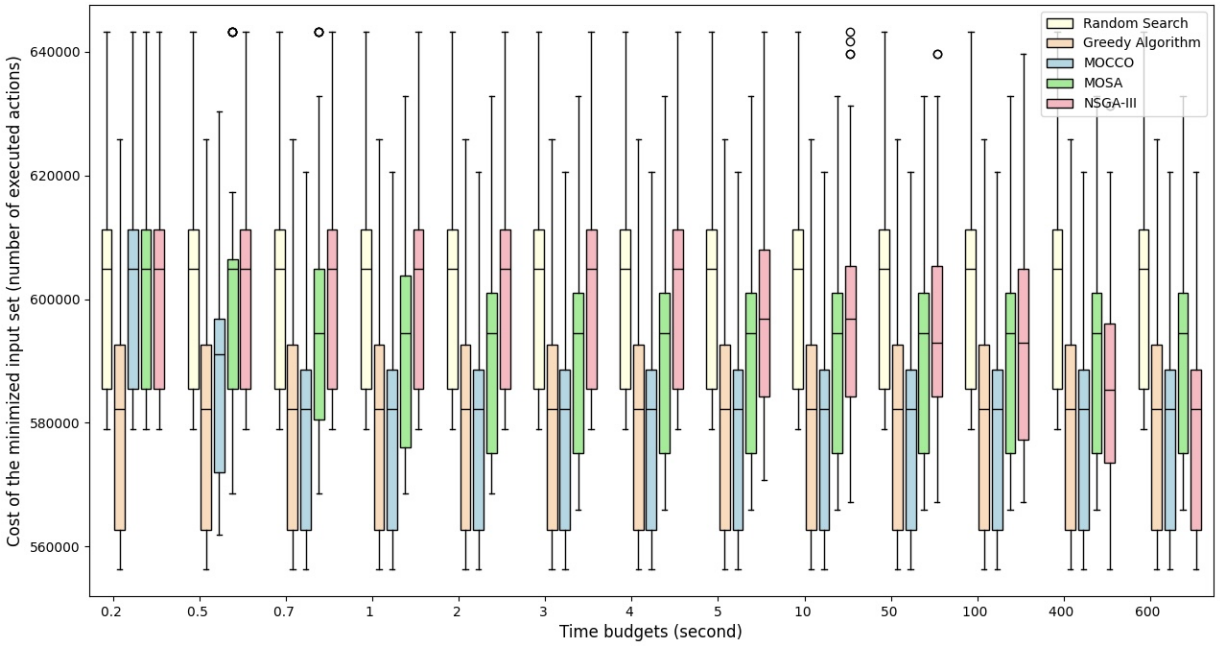

Figure 3.7: Joomla: Cost of the minimized input sets using Random Search, Greedy Algorithm, MOCCO, MOSA, and NSGA-III under different time budgets for 50 runs of BDK.

The results for Jenkins with a time budget of 600 seconds are detailed in Table 3.13. The small p-values and effect sizes observed in the MOCCO row indicates that **MOCCO obtained minimized input sets with significantly smaller costs than all the alternative approaches**. Moreover, the effect size for all baselines but the greedy algorithm is 0, indicating that the minimized input sets obtained by MOCCO consistently have smaller costs across all 50 runs. For the greedy algorithm, the effect size is 0.01 because it performs as well as MOCCO for a few runs (6 out of 50 runs). For Joomla, the results with a time budget of 400 and 600 seconds are respectively detailed in Tables 3.14 and 3.15. For all time budgets, the small p-values and the effect sizes of 0 indicate that **MOCCO performed better than random search and MOSA in all 50 runs**, but the results are more nuanced for the greedy algorithm and NSGA-III. For all time budgets, p-values indicate that **MOCCO performed significantly better than the greedy algorithm**. The latter managed to find the optimal solution for most runs (41 out of 50 runs), yielding an effect size of 0.34, which is still in favor of MOCCO. We conjecture this is because the greedy algorithm does not consider input block as individual objectives and hence, as opposed to MOCCO, does not take into account relevant information when selecting inputs (Section 3.3.2). For a 400-second time budget, the p-value indicates that **MOCCO performed significantly better than NSGA-III**, but NSGA-III managed to find the optimal solution for some runs, hence the effect size of 0.11, which is still in favor of MOCCO. For a 600-second time budget, the p-value is 1.0 and the effect size is 0.5, indicating that both approaches find the same results for all 50 runs, i.e., the optimal solutions. We conjecture that NSGA-III manages to achieve the same results as MOCCO for Joomla but not Jenkins because the former's reduced input set has fewer redundant

Table 3.13: Comparison of genetic algorithms for Jenkins (600 s).

costs		Random	Greedy	MOCCO	MOSA	NSGA-III
Random	p		1.8×10^{-15}	1.8×10^{-15}	1.8×10^{-15}	1.8×10^{-15}
	E		1.0	1.0	1.0	1.0
Greedy	p	1.8×10^{-15}		1.1×10^{-9}	1.8×10^{-15}	1.8×10^{-15}
	E	0.0		0.99	0.0	0.0
MOCCO	p	1.8×10^{-15}	1.1×10^{-9}		1.8×10^{-15}	1.8×10^{-15}
	E	0.0	0.01		0.0	0.0
MOSA	p	1.8×10^{-15}	1.8×10^{-15}	1.8×10^{-15}		1.8×10^{-15}
	E	0.0	1.0	1.0		1.0
NSGA-III	p	1.8×10^{-15}	1.8×10^{-15}	1.8×10^{-15}	1.8×10^{-15}	
	E	0.0	1.0	1.0	0.0	

Table 3.14: Comparison of genetic algorithms for Joomla (400 s).

costs		Random	Greedy	MOCCO	MOSA	NSGA-III
Random	p		1.8×10^{-15}	1.8×10^{-15}	1.8×10^{-15}	4.7×10^{-10}
	E		1.0	1.0	1.0	1.0
Greedy	p	1.8×10^{-15}		3.2×10^{-2}	1.8×10^{-15}	1.3×10^{-5}
	E	0.0		0.66	0.0	0.15
MOCCO	p	1.8×10^{-15}	3.2×10^{-2}		1.8×10^{-15}	9.4×10^{-7}
	E	0.0	0.34		0.0	0.11
MOSA	p	1.8×10^{-15}	1.8×10^{-15}	1.8×10^{-15}		1.8×10^{-5}
	E	0.0	1.0	1.0		0.84
NSGA-III	p	4.7×10^{-10}	1.3×10^{-5}	9.4×10^{-7}	1.8×10^{-5}	
	E	0.0	0.85	0.89	0.16	

inputs to be removed compared to Jenkins, making the problem computationally simpler. Since MOCCO was the only search algorithm able to consistently find the optimal solution for Jenkins for every time budget and since, for Joomla, MOCCO finds the optimal solution almost three orders of magnitude faster than the only baseline, i.e., NSGA-III, that manages to obtain the same results, we conclude that **MOCCO outperforms all baselines when accounting for both minimized input set cost and execution time**.

Nevertheless, the greedy algorithm finds approximations that are relatively close to those of MOCCO. To determine the significance of the difference in practice, we estimate how this difference in cost translates into a difference in execution time, based on our execution time results (Section 3.10.2). For Jenkins, it took 6119 minutes to execute a minimized input set of cost 119089 actions. Since the greedy algorithms obtained minimized inputs sets with on average 611.74 more actions and up to 1095 actions, this translates into a difference of 31.43 minutes on average to execute the MRs, up to 56.3 minutes in the worst case. For Joomla, the minimized input set of cost 582181 took 3675 minutes, hence given the difference between the greedy algorithm and MOCCO of 949.32 actions on average with a maximum of 5274 actions, this translates into a difference of 6 minutes on average, up to 33.29 minutes in the worst case. However, relatively to the total execution time, these differences may not have a practical impact. **Therefore, we conclude, from our case studies, that the greedy algorithm, despite its limitations, is a good alternative.**

Table 3.15: Comparison of genetic algorithms for Joomla (600 s).

costs		Random	Greedy	MOCCO	MOSA	NSGA-III
Random	p E		1.8×10^{-15} 1.0	1.8×10^{-15} 1.0	1.8×10^{-15} 1.0	1.8×10^{-15} 1.0
Greedy	p E	1.8×10^{-15} 0.0		3.2×10^{-2} 0.66	1.8×10^{-15} 0.0	3.2×10^{-2} 0.66
MOCCO	p E	1.8×10^{-15} 0.0	3.2×10^{-2} 0.34		1.8×10^{-15} 0.0	1.0 0.5
MOSA	p E	1.8×10^{-15} 0.0	1.8×10^{-15} 1.0	1.8×10^{-15} 1.0		1.8×10^{-15} 1.0
NSGA-III	p E	1.8×10^{-15} 0.0	3.2×10^{-2} 0.34	1.0 0.5	1.8×10^{-15} 0.0	

3.11 Threats to Validity

In this section, we discuss internal, conclusion, construct, and external validity according to conventional practices [239].

3.11.1 Internal Validity

A potential internal threat concerns inadequate data pre-processing, which may adversely impact our results. Indeed, clustering relies on the computed similarity among the pre-processed outputs and inputs. To address this potential concern, we have conducted a manual investigation of the quality of the clusters obtained without pre-processing. This led us to remove, from the textual content extracted from each Web page, all the content that was shared by many Web pages, like system version, date, or (when present) the menu of the Web page.

For RQ1 on vulnerability detection, one potential threat we face is missing inputs that would be able to exercise a vulnerability or incorrectly considering that an input is able to exercise a vulnerability. To ensure our list of inputs triggering vulnerabilities is complete, one author inspected all the *MST-wi* execution logs to look for failures.

3.11.2 Conclusion Validity

For RQ2, we rely on a non-parametric test (i.e., Mann-Whitney-Wilcoxon test) to evaluate the statistical and practical significance of differences in results, computing p and Vargha and Delaney's A_{12} metric for effect size. Moreover, to deal with the randomness inherent to search algorithms, all the configurations and baselines were executed over 50 runs.

Randomness may also arise from (1) the workload of the machines employed to conduct experiments, potentially slowing down the performance of *MST-wi*, AIM, and the case study subjects, and (2) the presence of other users interacting with the software under test, which can impact both execution time and system outputs. To address these concerns, we conducted experiments in dedicated environments, ensuring that the study subjects were exclusively utilized by AIM.

3.11.3 Construct Validity

The constructs considered in our work are vulnerability detection effectiveness and input set reduction effectiveness. Vulnerability detection effectiveness is measured in terms of vulnerability detection rate. Reduction effectiveness is measured in terms of MR execution time, size and cost of the minimized input set, and AIM execution time for each configuration. As it is expensive to execute all 18 configurations on the MRs, we consider the size of the input set and its cost to select the most efficient configuration. The cost of the input set has been defined in Section 3.3.1 and shown to be linearly correlated with MR execution time, thus enabling us to evaluate the efficiency of the results.

Finally, we executed the minimized input set obtained from the best configuration on the MRs and compared the obtained execution time, plus the AIM execution time required to minimize the initial input set, with the MRs execution time obtained with the initial input set. Execution time is a direct measure, allowing us to evaluate whether, for systems akin to our case study subjects, AIM should be adopted for making vulnerability testing more efficient and scalable.

3.11.4 External Validity

One threat to the generalizability of our results stems from the benchmark that we used. It includes 160 inputs for Jenkins and 148 inputs for Joomla. Furthermore, we considered the list of vulnerabilities in Jenkins and Joomla that were successfully triggered with *MST-wi*. However, even if in this study we used *MST-wi* to collect our data, the AIM approach does not depend on a particular data collector, and using or implementing another data collector would enable the use of our approach with other frameworks. Moreover, even if we relied on previously obtained MRs to be sure they detect vulnerabilities in the considered Web systems, AIM is a general approach for metamorphic security testing which does not depend on the considered MRs. Finally, in Section 3.10.1, we highlighted that the different input/output interfaces provided by Jenkins and Joomla, along with the diverse types of vulnerabilities they contain, is in support of the generalizability of our results. Furthermore, the AIM approach can be generalized to other Web systems, if the data collection and pre-processing components are updated accordingly. Nevertheless, further studies involving systems with known vulnerabilities are needed.

3.12 Related Work

MT enables the execution of a SUT with a potentially infinite set of inputs thus being more effective than testing techniques requiring the manual specification of either test inputs or oracles. However, in MT, the manual definition of metamorphic relations (MRs) is an expensive activity because it requires that engineers first acquire enough information on the subject under test and then analyze the testing problem to identify MRs. For this reason, in the past, researchers focused on both the definition of methodologies supporting the

identification of MRs [68,220] and the development of techniques for the automated generation of MRs, based on meta-heuristic search [42,251] and natural language processing [52], and targeting query-based systems [205] and Cyber-Physical Systems [42].

However, source inputs also impact the effectiveness and performance of MT; indeed, MRs generate follow-up inputs from source inputs and both are executed by the SUT. Consequently, the research community has recently shown increasing interest towards investigating the impact of source inputs on MT. We summarize the most relevant works in the following paragraphs. Note that all these studies focus on general fault detection, while we focus on metamorphic security testing for Web systems. However, our approach could also be applied to fault detection while the approaches below could also be applied to security testing. We therefore compare these approaches without considering their difference in application. However, we excluded from our survey those approaches that study the effect of source and follow-up inputs on the metamorphic testing of systems that largely differ from ours (i.e., sequence alignment programs [226], system validation [253], and deep neural networks [255]). In the following paragraphs, we group the surveyed works into three categories: input generation techniques, input selection techniques, and feedback-directed metamorphic testing.

Input generation techniques for MT use white-box approaches based on knowledge of the source code (mainly, for statement or branch coverage), while we use a black-box approach based on input and output information (Section 3.6). For instance, a study [201] leveraged the evolutionary search approach EvoSuite [86] to evolve whole input sets in order to obtain inputs that lead to more branch coverage or to different results on the mutated and non-mutated versions of the source code. Another example study [221] leveraged symbolic execution to collect constraints of program branches covered by execution paths, then solved these constraints to generate the corresponding source inputs. Finally, the execution of the generated inputs on the SUT was prioritized based on their contribution regarding uncovered statements. In this case, both generation and prioritization phases were white-box approaches based on branch coverage. Note that, while our approach on input set minimization could be seen as similar to input prioritization, both studies focused on increasing coverage, while we focused on reducing cost while maintaining full coverage.

Input selection techniques share the same objective of our work (i.e., reducing the number of source inputs while maximizing MT effectiveness). Because of its simplicity, random testing (RT) is a common strategy for test suite minimization [236, 252] that has been used in MT [97]. RT was enhanced with Adaptive Random Testing (ART), a technique for obtaining source inputs spread across the input domain with the aim of finding failures with fewer number of inputs than RT. As input selection technique for MT, ART outperforms RT in terms of fault detection [45], which (as in the following studies) was evaluated using the F-measure, i.e., the number of inputs necessary to reveal the first failure. In the AIM approach, our action clustering step (Section 3.6.3) bears similarities with ART, since we partition inputs based on action parameters which are relevant for our SUT. But, instead of assuming that close inputs lead to close outputs, we directly used SUT outputs during our output clustering step (Section 3.6.2), since they are inexpensive to obtain compared to executing MRs. Finally, instead of only counting the number of inputs as in the F-measure (or the size of the input set, in the context of input generation),

we considered the cost of each source input as the number of executed actions as surrogate measure, which is tailored to reducing MR execution time in the context of Web systems. Instead of focusing only on distances between source inputs as in ART, another study [97] also investigated distances with follow-up inputs, which is an improvement since usually there are more follow-up inputs than source inputs. This led to the Metamorphic testing-based adaptive random testing (MT-ART) technique, which performed better than other ART algorithms regarding test effectiveness, test efficiency, and test coverage (considering statement, block, and branch coverage). Unfortunately, in the AIM approach, we could not consider follow-up inputs to drive the input selection, since executing MRs to generate these follow-up inputs would defeat our purpose of reducing MR execution time.

Finally, while studies on MT usually focus either on the identification of effective MRs or on input generation/selection, a recent study proposed *feedback-directed metamorphic testing* (FDMT) [219] to determine the next test to perform (both in terms of source input and MR), based on previous test results. They proposed adaptive partition testing (APT) to dynamically select source inputs, based on input categories that lead to fault detection, and a diversity-oriented strategy for MR selection (DOMR) to select an MR generating follow-up inputs that are as different as possible from the already obtained ones. While this approach is promising in general, it is not adapted to our case, where we consider a fixed set of MRs, MR selection being considered outside the scope of this thesis. Moreover, since we aim to reduce MR execution time, we cannot execute them and use execution information to guide source input or MR selection during testing. Finally, in our problem definition (Section 3.3), we do not consider source inputs independently from each other, which is why we reduced (Section 3.7) then minimized (Section 3.8) the cost of the input set as a whole.

3.13 Conclusion

As demonstrated in our previous work [47], metamorphic testing alleviates the oracle problem for the automated security testing of Web systems. However, metamorphic testing has shown to be a time-consuming approach. Our approach (AIM) aims to reduce the cost of metamorphic security testing by minimizing the initial input set while preserving its capability at exercising vulnerabilities. Our contributions include 1) a clustering-based black box approach that identifies similar inputs based on their security properties, 2) IMPRO, an approach to reduce the search space as much as possible, then divide it into smaller independent parts, 3) MOCCO, a novel genetic algorithm which is able to efficiently select diverse inputs while minimizing their total cost, and 4) a testing framework automatically performing input set minimization.

We considered 18 different configurations for AIM and we evaluated our approach on two open-source Web systems, Jenkins and Joomla, in terms of vulnerability detection rate and magnitude of the input set reduction. Our empirical results show that the best configuration for AIM is BDK: Bag distance, DBSCAN to cluster the outputs, and K-means to cluster the inputs. The results show that our approach can automatically reduce MRs execution time by 84% for Jenkins and 82% for Joomla while preserving full vulnerability

detection. Across 50 runs, the BDK configuration consistently detected all vulnerabilities in Jenkins and Joomla. We also compared AIM with four baselines common in security testing. Notably, none of the baselines reached full vulnerability coverage. Among them, AK (ART baseline using K-means) emerged as the closest to achieving full vulnerability coverage. All AIM configurations with full vulnerability coverage outperformed this baseline in terms of minimized input set size and cost, demonstrating the effectiveness of our approach in reducing the cost of metamorphic security testing.

Furthermore, we compared the effectiveness of MOCCO, in terms of minimized input set cost and execution time, with four other search algorithms. The results on Jenkins showed that MOCCO obtained minimized input sets with significantly lower costs than all the alternative approaches. The only baseline that could find the optimal solutions on Joomla was NSGA-III, though MOCCO did so almost three orders of magnitude faster. Among the considered alternative search algorithms, greedy was the only algorithm that consistently found results close to those of MOCCO, on both Jenkins and Joomla. Therefore, we conclude that MOCCO outperforms all baselines in terms of minimized input set cost and execution time, while the greedy algorithm, despite its theoretical limitations, remains a viable alternative in practice.

Chapter 4

Thesis Conclusion

This thesis presents a comprehensive solution for the automated security testing of Web systems, focusing on detecting a broad range of vulnerabilities and improving the scalability of testing tools by minimizing input sets. Given the importance of security testing in critical domains, such as e-commerce and online banking, ensuring the effectiveness of security testing methods in terms of number of detected vulnerabilities and false positive rate is crucial. This research addresses key challenges, including the oracle problem in automated security testing and the scalability issue of security testing methods for real-world applications. The primary contributions of this thesis are as follows:

- **Proposing an automated Metamorphic Security Testing Approach:** This thesis addresses a wide range of vulnerabilities, i.e., 102 known vulnerability types of Web systems listed in the Common Weaknesses Enumeration Repository [26] and OWASP [10]. This approach utilizes MRs to capture security properties of Web systems and automatically detect vulnerabilities based on those relations. The empirical results with two open-source Web systems, Jenkins and Joomla, showed that the approach required limited manual effort and detected 85.71% of the targeted vulnerabilities, thus suggesting it is highly effective. Moreover, since at most 0.19% of the executed follow-up inputs led to a false alarm, it is also very efficient.
- **Developing *MST-wi* Framework:** This research also presented *MST-wi*, a testing framework for MST that automatically performs security testing based on the MRs and the collected data. *MST-wi* is built on top of the following novel solutions:
  1. The Security Metamorphic Relation Language (SMRL), a Domain-Specific Language (DSL) for specifying MRs for software security testing. SMRL is supported by a custom editor, implemented as a plug-in for the Eclipse IDE [1], which facilitates the specification of MRs in practice.
  2. A catalog of 76 system-agnostic MRs targeting 102 known security vulnerabilities of Web systems.
  3. A data collection framework that automatically collects the data required to perform MT.

- **Presenting an Input Set Minimization Approach:** This thesis proposed AIM, a novel approach to minimize input sets for metamorphic testing while retaining inputs able to detect vulnerabilities. AIM utilizes a clustering-based black-box approach to group similar inputs based on their security characteristics and incorporates a novel genetic algorithm to efficiently select diverse inputs while reducing overall cost. The results showed that our approach can automatically reduce MRs execution time by 84% for Jenkins and 82% for Joomla, while preserving full vulnerability detection.
- **Developing the AIM Framework:** This research presented a prototype framework for AIM, automating the process of input set minimization for Web systems. This tool includes the following novel components:

  1. An extension of the *MST-wi* framework to retrieve output data and extract cost information about MRs without executing them.
  2. A black-box approach leveraging clustering algorithms to partition the initial input set based on security-related characteristics in order to keep a small number of representative inputs.
  3. MOCCO, a novel genetic algorithm specifically designed to fit our problem and which is able to efficiently select diverse inputs while minimizing their total cost.
  4. IMPRO, an approach to reduce the search space to its minimal extent, and then divide it in smaller, easier to minimize, independent parts.

4.1 Future Research Directions

The results of this thesis open several avenues for future research:

- **Test case prioritization:** A future study could explore a test case prioritization technique that facilitates earlier vulnerability detection by prioritizing inputs from the minimized input set obtained with AIM, according to their likelihood of detecting vulnerabilities.
- **Using LLM models to automatically generate MRs:** Since defining MRs is currently done manually and requires domain expertise, future study could explore using LLMs and prompt engineering techniques to automate MR definition based on the description of a given weakness from the Common Weaknesses Enumeration Repository [26] and a list of *MST-wi*'s APIs.
- **Extending *MST-wi* to cover a larger set of weaknesses:** A future research direction could involve extending the set of MRs to cover a wider range of weaknesses in Web systems, such as those that require server-side access, e.g., Server-Side Request Forgery (SSRF) [19], or the ones that can be discovered only by controlling a third-party system, e.g., validation of certificate with host mismatch [16]. Additionally, since *MST-wi*'s effectiveness largely depends on the capability of the Web

crawler used to gather source inputs, future research should focus on developing Web crawlers that mimic end-user behavior and can explore a wide range of features in the system under test.

- **Employing MST for Network Security testing:** Future research could extend the application of MST to network security testing by adapting its capabilities to identify network-specific vulnerabilities, such as misconfigurations, unauthorized access points, and potential DoS attack vectors. Expanding MST into this domain would offer more comprehensive security guarantees, ultimately contributing to stronger defenses for network infrastructures and critical applications.

- **Extending MST as A DevSecOps Tool:** Given advances in the industry and the growing adoption of DevSecOps practices to enhance software security, a promising future research direction could involve extending *MST-wi* 's features for integration into the DevSecOps pipeline.

4.2 Final Remarks

In conclusion, this thesis has introduced and demonstrated innovative approaches in automated security testing, particularly for Web systems, contributing to the advancement of effective and scalable testing methods. By addressing key challenges in vulnerability detection, input set minimization, and adaptability to real-world applications, this work lays a foundation for future research and development in the field. The proposed *MST-wi* and AIM frameworks not only push the boundaries of metamorphic testing but also highlight the importance of integrating security testing into diverse and evolving technological landscapes. As cybersecurity threats continue to grow, the insights and tools presented here offer promising directions for building robust, efficient, and resilient security testing solutions.

References

[1] Eclipse IDE, https://www.eclipse.org/ide/, 2018.

[2] Joomla, https://www.joomla.org/, 2018.

[3] Xtext, https://www.eclipse.org/Xtext/, 2018.

[4] JUnit, https://junit.org/, 2019.

[5] CWE Top 25 Most Dangerous Software Errors. https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html, 2021.

[6] CWE VIEW: Architectural Concepts. https://cwe.mitre.org/data/definitions/1008.html, 2021.

[7] CWE VIEW: Hardware Design. https://cwe.mitre.org/data/definitions/1194.html, 2021.

[8] CWE VIEW: Research Concepts. https://cwe.mitre.org/data/definitions/1000.html, 2021.

[9] CWE VIEW: Software Development Concepts. https://cwe.mitre.org/data/definitions/699.html, 2021.

[10] OWASP Top 10 Mobile Security Risks. https://www.owasp.org/index.php/Mobile_Top_10_2021-Top_10, 2021.

[11] OWASP Top 10 Web Application Security Risks. https://owasp.org/www-project-top-ten/, 2021.

[12] ASM bytecode manipulation framework. https://asm.ow2.io/, 2022.

[13] Common attack pattern enumeration and classification (CAPEC). https://capec.mitre.org, 2022.

[14] CWE - Common Weakness Enumeration. https://cwe.mitre.org, 2022.

[15] CWE-287: Improper authentication. https://cwe.mitre.org/data/definitions/287.html, 2022.

[16] CWE-297: Improper validation of certificate with host mismatch. https://cwe.mitre.org/data/definitions/297.html, 2022.

[17] CWE-613: Insufficient session expiration. https://cwe.mitre.org/data/definitions/613.html, 2022.

[18] CWE-640: Weak password recovery mechanism for forgotten password. https://cwe.mitre.org/data/definitions/640.html, 2022.

[19] CWE-918: Server-side request forgery (ssrf). https://cwe.mitre.org/data/definitions/918.html, 2022.

[20] CWE Category: sert cei c coding standards. https://cwe.mitre.org/data/definitions/1154.html, 2022.

[21] CWE Category: software fault patterns. https://cwe.mitre.org/data/definitions/888.html, 2022.

[22] CWE-FAQ. https://cwe.mitre.org/about/faq.html#A.1, 2022.

[23] CWE View: Weaknesses in owasp top ten (2017). https://cwe.mitre.org/data/definitions/1026.html, 2022.

[24] Format string attack. https://owasp.org/www-community/attacks/Format_string_attack, 2022.

[25] Java Platform, Enterprise Edition. https://www.oracle.com/java/technologies/java-ee-glance.html, 2022.

[26] MITRE common weaknesses enumeration project, which provides a taxonomy of vulnerability types. https://cwe.mitre.org, 2022.

[27] Open Web Application Security Project. https://www.owasp.org/, 2022.

[28] WSTG - v4.1: Testing Session Timeout. https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Security_Testing/06-Session_Management_Testing/07-Testing_Session_Timeout.html, 2022.

[29] Zohreh Aghababaeyan, Manel Abdellatif, Mahboubeh Dadkhah, and Lionel Briand. Deepgd: A multi-objective black-box test selection approach for deep neural networks. *ACM Trans. Softw. Eng. Methodol.*, 33(6), jun 2024.

[30] Christopher Alexander. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[31] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, Cambridge, 2016.

[32] Apache software foundation. Apache web server, 2022. https://httpd.apache.org/.

[33] Dennis Appelt, Nadia Alshahwan, and Lionel Briand. Assessing the impact of firewalls and database proxies on sql injection testing. In *FITTEST'13*, pages 32–47, 2013.

[34] Dennis Appelt, Cu Duy Nguyen, Lionel C. Briand, and Nadia Alshahwan. Automated testing for sql injection vulnerabilities: An input mutation approach. In *ISSTA'14*, pages 259–269, 2014.

[35] Andrea Arcuri. It does matter how you normalise the branch distance in search based software testing. In *Third International Conference on Software Testing, Verification and Validation*, pages 205–214, Paris, France, April 2010. IEEE.

[36] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Softw. Test. Verif. Reliab.*, 24(3):219–250, May 2014.

[37] Chetan Arora, Mehrdad Sabetzadeh, Lionel Briand, and Frank Zimmer. Automated extraction and clustering of requirements glossary terms. *IEEE Transactions on Software Engineering*, 43(10):918–945, 2017.

[38] Chittineni Aruna and R Siva Ram Prasad. Metamorphic relations to improve the test accuracy of multi precision arithmetic software applications. In *ICACCI'14*, pages 2244–2248, 2014.

[39] Mohammed Attaoui, Hazem Fahmy, Fabrizio Pastore, and Lionel Briand. Black-box safety analysis and retraining of dnns based on feature extraction and clustering. *ACM Trans. Softw. Eng. Methodol.*, 32(3), Apr 2023.

[40] Mohammed Oualid Attaoui, Hazem Fahmy, Fabrizio Pastore, and Lionel Briand. Dnn explanation for safety analysis: an empirical evaluation of clustering-based approaches. *arXiv*, 2023.

[41] Andrew Austin, Casper Holmgreen, and Laurie Williams. A comparison of the efficiency and effectiveness of vulnerability discovery techniques. *Information and Software Technology*, 55(7):1279–1288, 2013.

[42] Jon Ayerdi, Valerio Terragni, Aitor Arrieta, Paolo Tonella, Goiuria Sagardui, and Maite Arratibel. Generating metamorphic relations for cyber-physical systems with genetic programming: An industrial case study. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1264–1274, New York, NY, USA, 2021. Association for Computing Machinery.

[43] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[44] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. String matching with metric trees using an approximate distance. In *String Processing and Information Retrieval: 9th International Symposium, SPIRE 2002 Lisbon, Portugal, September 11–13, 2002 Proceedings 9*, pages 271–283, Berlin, Heidelberg, 2002. Springer.

[45] Arlinta Christy Barus, Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, and Heinz W. Schmidt. The impact of source test case selection on the effectiveness of metamorphic testing. In *Proceedings of the 1st International Workshop on Metamorphic Testing*, MET '16, page 5–11, New York, NY, USA, 2016. Association for Computing Machinery.

[46] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the art: Automated black-box web application vulnerability testing. In *SP'10*, pages 332–345, 2010.

[47] Nazanin Chaleshtari Bayati, Fabrizio Pastore, Arda Goknil, and Lionel C. Briand. Metamorphic testing for web system security. *IEEE Transactions on Software Engineering*, 49(6):3430–3471, 2023.

[48] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. Finding software vulnerabilities by smart fuzzing. In *ICST'11*, pages 427–430, 2011.

[49] Nikolai Mansourov Ben A. Calloni, Djenana Campana. 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, 2011. `https://apps.dtic.mil/docs/citations/ADB381215`.

[50] Antonia Bertolino, Said Daoudagh, Francesca Lonetti, Eda Marchetti, Fabio Martinelli, and Paolo Mori. Testing of PolPA authorization systems. In *AST'12*, pages 8–14, 2012.

[51] Matteo Biagiola, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Diversity-based web test generation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 142–153, New York, NY, USA, 2019. Association for Computing Machinery.

[52] Arianna Blasi, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Antonio Carzaniga. Memo: Automatically identifying metamorphic relations in javadoc comments for test automation. *Journal of Systems and Software*, 181:111041, 2021.

[53] Abian Blome, Martin Ochoa, Keqin Li, Michele Peroli, and Mohammad Torabi Dashti. Vera: A flexible model-based vulnerability testing tool. In *ICST'13*, pages 471–478, 2013.

[54] Stanislav Busygin, Oleg Prokopyev, and Panos M. Pardalos. Biclustering in data mining. *Computers & Operations Research*, 35(9):2964–2987, 2008. Part Special Issue: Bio-inspired Methods in Combinatorial Optimization.

[55] Nazanin Bayati Chaleshtari, Phu X. Mai, Fabrizio Pastore, Arda Goknil, and Lionel C. Briand. SMRL editor executable, catalog of MRs, MT framework, experimental data. `https://sntsvv.github.io/SMRL/`, 2022.

[56] Nazanin Bayati Chaleshtari, Yoann Marquer, Fabrizio Pastore, and Lionel C. Briand. Replication package, 2024. https://zenodo.org/records/10661329.

[57] Nazanin Bayati Chaleshtari, Fabrizio Pastore, Arda Goknil, and Lionel C. Briand. Replicability package, 2023. https://doi.org/10.5281/zenodo.7702754.

[58] Wing Kwong Chan, Tsong Y Chen, Shing Chi Cheung, TH Tse, and Zhenyu Zhang. Towards the testing of power-aware software applications for wireless sensor networks. In *ADA Europe'07*, pages 84–99, 2007.

[59] Wing Kwong Chan, Shing Chi Cheung, and Karl RPH Leung. A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Services Research*, 4(2):61–81, 2007.

[60] Wing Kwong Chan, Jeffrey CF Ho, and TH Tse. Finding failures from passed test cases: Improving the pattern classification approach to the testing of mesh simplification programs. *Software Testing, Verification and Reliability*, 20(2):89–120, 2010.

[61] WK Chan, Jeffrey CF Ho, and TH Tse. Piping classification to metamorphic testing: An empirical study towards better effectiveness for the identification of failures in mesh simplification programs. In *COMPSAC'07*, volume 1, pages 397–404, 2007.

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

[63] Tsong Yueh Chen, Shing-Chi Cheung, and Siu-Ming Yiu. Metamorphic testing: a new approach for generating next test cases. Technical report, The Hong Kong University of Science and Technology, 1998.

[64] Tsong Yueh Chen, Jianqiang Feng, and TH Tse. Metamorphic testing of programs on partial differential equations: a case study. In *COMPSAC'02*, pages 327–333, 2002.

[65] Tsong Yueh Chen, Joshua WK Ho, Huai Liu, and Xiaoyuan Xie. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC bioinformatics*, 10(1):24, 2009.

[66] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challanges and opportunities. *ACM Computing Surveys*, 51(1), 2018.

[67] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, and Shengqiong Wang. Conformance testing of network simulators based on metamorphic testing technique. In *FORTE'09*, pages 243–248, 2009.

[68] Tsong Yueh Chen, Pak-Lok Poon, and Xiaoyuan Xie. Metric: Metamorphic relation identification based on the category-choice framework. *Journal of Systems and Software*, 116:177–190, 2016.

[69] Emilio Cruciani, Breno Miranda, Roberto Verdecchia, and Antonia Bertolino. Scalable approaches for test suite reduction. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 419–429, May 2019.

[70] CVE, MITRE Corporation. Common vulnerabilities and exposures. `https://cve.mitre.org/cve/`, 2018.

[71] Dani Deahl. Another Facebook Vulnerability. `https://bit.ly/3gtPo80`, 2019.

[72] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, April 2002.

[73] Kalyanmoy Deb and Himanshu Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints. *IEEE Transactions on Evolutionary Computation*, 18(4):577–601, Aug 2014.

[74] Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *J. Mach. Learn. Res.*, 7:1–30, dec 2006.

[75] Junhua Ding, Tong Wu, Dianxiang Wu, Jun Q Lu, and Xin-Hua Hu. Metamorphic testing of a monte carlo modeling program. In *AST'11*, pages 1–7, 2011.

[76] Irit Dinur and David Steurer. Analytical approach to parallel repetition. In *Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '14, page 624–633, New York, NY, USA, 2014. Association for Computing Machinery.

[77] Eclipse Foundation. Jenkins ci/cd server. `https://jenkins.io/`, 2018.

[78] Eclipse Foundation. Jetty application server. `https://www.eclipse.org/jetty/`, 2022.

[79] EDLAH2: Active and Assisted Living Programme. `http://www.edlah2.eu/`, 2018.

[80] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. Xbase: Implementing domain-specific languages for java. *ACM SIGPLAN Notices - GPCE '12*, 48(3):112–121, 2012.

[81] Sarah Elder, Nusrat Zahan, Rui Shu, Monica Metro, Valeri Kozarev, Tim Menzies, and Laurie Williams. Do I really need all this work to find vulnerabilities? *Empirical Software Engineering*, 27(6):154, 2022.

[82] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231, Portland, Oregon, 1996. AAAI Press.

[83] Michael Felderer, Berthold Agreiter, Philipp Zech, and Ruth Breu. A classification for model-based security testing. In *VALID'11*, pages 109–114, 2011.

[84] Michael Felderer, Matthias Buchler, Martin Johns, Achim D. Brucker, Ruth Breu, and Alexander Pretschner. Security testing: A survey. *Advances in Computers*, 101:1–51, 2016.

[85] Michael Felderer, Philipp Zech, Ruth Breu, Matthias Büchler, and Alexander Pretschner. Model-based security testing: A taxonomy and systematic classification. *Software: Testing, Verification and Reliability*, 26(2):119–148, 2016.

[86] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 416–419, New York, NY, USA, 2011. Association for Computing Machinery.

[87] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, Feb 2013.

[88] Vahid Garousi, Michael Felderer, and Feyza Nur Kılıçaslan. A survey on software testability. *Information and Software Technology*, 108:35–64, 2019.

[89] Ralph Guderlei and Johannes Mayer. 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, 2007.

[90] Neha Gupta, Arun Sharma, and Manoj Kumar Pachariya. Multi-objective test suite optimization for detection and localization of software faults. *Journal of King Saud University - Computer and Information Sciences*, 34(6, Part A):2897–2909, 2022.

[91] Charles Haley, Robin Laney, Jonathan Moffett, and Bashar Nuseibeh. Security requirements engineering: A framework for representation and analysis. *IEEE Transactions on Software Engineering*, 34(1):133–153, 2008.

[92] Marshall Jr. Hall. *The Theory of Groups*. MacMillan, USA, 1959.

[93] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *USENIX Security'13*, pages 49–64, 2013.

[94] Ke He, Zhiyong Feng, and Xiaohong Li. An attack scenario based approach for software security testing at design stage. In *ISCSCT'08*, pages 782–787, 2008.

[95] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(1):1–42, 2013.

[96] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. Web application security assessment by fault injection and behavior monitoring. In *WWW'03*, pages 148–159, 2003.

[97] Zhan-wei Hui, Xiaojuan Wang, Song Huang, and Sen Yang. MT-ART: A Test Case Generation Method Based on Adaptive Random Testing and Metamorphic Relation. *IEEE Transactions on Reliability*, 70(4):1397–1421, Dec 2021.

[98] Himanshu Jain and Kalyanmoy Deb. An evolutionary many-objective optimization algorithm using reference-point based nondominated sorting approach, part ii: Handling constraints and extending to an adaptive approach. *IEEE Transactions on Evolutionary Computation*, 18(4):602–622, Aug 2014.

[99] Tahir Jameel, Mengxiang Lin, and Liu Chao. Test oracles based on metamorphic relations for image processing applications. In *SNPD'15*, pages 1–6, 2015.

[100] Mingyue Jiang, Tsong Yueh Chen, Fei-Ching Kuo, and Zuohua Ding. Testing central processing unit scheduling algorithms using metamorphic testing. In *ICSESS'13*, pages 530–536, 2013.

[101] Jan Jürjens. UMLsec: Extending UML for secure systems development. In *UML'02*, pages 412–425, 2002.

[102] Jan Jürjens. *Secure Systems Development with UML*. Springer Science & Business Media, 2005.

[103] Jan Jürjens. Sound methods and effective tools for model-based security engineering with UML. In *ICSE'05*, pages 322–331, 2005.

[104] Jan Jürjens. Model-based security testing using UMLsec: A case study. *Electronic Notes in Theoretical Computer Science*, 220(1):93–104, 2008.

[105] René Just and Franz Schweiggert. Evaluating testing strategies for imaging software by means of mutation analysis. In *ICSTW'09*, pages 205–209, 2009.

[106] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. SecuBat: A web vulnerability scanner. In *WWW'06*, pages 247–246, 2006.

[107] Dave S. Kerby. The simple difference formula: An approach to teaching nonparametric correlation. *Comprehensive Psychology*, 3:11.IT.3.1, 2014.

[108] Mifa Kim, Tomoyuki Hiroyasu, Mitsunori Miki, and Shinya Watanabe. Spea2+: Improving the performance of the strength pareto evolutionary algorithm 2. In Xin Yao, Edmund K. Burke, José A. Lozano, Jim Smith, Juan Julián Merelo-Guervós, John A.

Bullinaria, Jonathan E. Rowe, Peter Tiňo, Ata Kabán, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VIII*, pages 742–751, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[109] Ayesha Kiran, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. A comprehensive investigation of modern test suite optimization trends, tools and techniques. *IEEE Access*, 7:89093–89117, 2019.

[110] Barbara Kitchenham, Lech Madeyski, David Budgen, Jacky Keung, Pearl Brereton, Stuart Charters, Shirley Gibbs, and Amnart Pohthong. Robust statistical methods for empirical software engineering. *Empirical Softw. Engg.*, 22(2):579–630, apr 2017.

[111] Bernhard Korte and Rainer Schrader. On the existence of fast approximation schemes. In Olvi L. Mangasarian, Robert R. Meyer, and Stephen M. Robinson, editors, *Nonlinear Programming 4*, pages 415–437. Academic Press, Madison, Wisconsin, 1981.

[112] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer Publishing Company, Incorporated, 6th edition, 2018.

[113] F-C Kuo, Zhiquan Q Zhou, Jun Ma, and Guangquan Zhang. Metamorphic testing of decision support systems: a case study. *IET software*, 4(4):294–301, 2010.

[114] Fei-Ching Kuo, Tsong Yueh Chen, and Wing K Tam. Testing embedded software by metamorphic testing: A wireless metering system case study. In *LCN'11*, pages 291–294, 2011.

[115] Fei-Ching Kuo, Shuang Liu, and Tsong Yueh Chen. Testing a binary space partitioning algorithm with metamorphic testing. In *SAC'11*, pages 1482–1489, 2011.

[116] David Lane. *Online Statistics Education: A Multimedia Course of Study*. Rice University, 2003.

[117] Andrei Lascu, Alastair F. Donaldson, Tobias Grosser, and Torsten Hoefler. Metamorphic fuzzing of c++ libraries. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 35–46, 2022.

[118] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM SIGPLAN Notices*, 49(6):216–226, 2014.

[119] Yves Le Traon, Tejeddine Mouelhi, and Benoit Baudry. Testing security policies: Going beyond functional testing. In *ISSRE'07*, pages 93–102, 2007.

[120] Franck Lebeau, Bruno Legeard, Fabien Peureux, and Alexandre Vernotte. Model-based vulnerability testing for web applications. In *ICSTW'13*, pages 445–452, 2013.

[121] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10, February 1966.

[122] Bingdong Li, Jinlong Li, Ke Tang, and Xin Yao. Many-objective evolutionary algorithms: A survey. *ACM Comput. Surv.*, 48(1), sep 2015.

[123] Zheng Li, Mark Harman, and Robert M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, April 2007.

[124] Huai Liu, Fei-Ching Kuo, Dave Towey, and Tsong Yueh Chen. How effectively does metamorphic testing alleviate the oracle problem? *IEEE Transactions on Software Engineering*, 40(1):4–22, 2014.

[125] Roberto Suggi Liverani. Integration of burp suite and crawljax. `https://github.com/portswigger/burp-csj`, 2022.

[126] Quang-Hung Luu, Man F Lau, Sebastian PH Ng, and Tsong Yueh Chen. Testing multiple linear regression systems with metamorphic testing. *Journal of Systems and Software*, 182:111062, 2021.

[127] Phu X. Mai. CVE-2020-2162: Stored XSS vulnerability in file parameters. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-2162`, 2020.

[128] Phu X. Mai, Arda Goknil, Fabrizio Pastore, and Lionel C. Briand. SMRL: a metamorphic security testing tool for web systems. In *ICSE (Companion Volume)'20*, pages 9–12, 2020.

[129] Phu X. Mai, Arda Goknil, 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*, 100:165–182, 2018.

[130] Phu X. Mai, Fabrizio Pastore, Arda Goknil, and Lionel C. Briand. A natural language programming approach for requirements-based security testing. *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 58–69, 2018.

[131] Phu X Mai, Fabrizio Pastore, Arda Goknil, and Lionel C Briand. Mcp: a security testing tool driven by requirements. In *ICSE (Companion Volume)'19*, pages 55–58, 2019.

[132] Phu X. Mai, Fabrizio Pastore, Arda Goknil, and Lionel C. Briand. Metamorphic security testing for web systems. In *ICST'20*, pages 186–197, 2020.

[133] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019.

[134] Aaron Marback, Hyunsook Do, Ke He, Samuel Kondamarri, and Dianxiang Xu. A threat model-based approach to security testing. *Software: Practice and Experience*, 43(2):241–258, 2013.

[135] Evan Martin and Tao Xie. Automated test generation for access control policies via change-impact analysis. In *SESS'07*, 2007.

[136] Evan Martin and Tao Xie. A fault model and mutation testing of access control policies. In *WWW'07*, pages 667–676, 2007.

[137] Evan Martin, Tao Xie, and Ting Yu. Defining and measuring policy coverage in testing access control policies. In *ICICS'06*, pages 139–158, 2006.

[138] Michael Martin and Monica S Lam. Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In *USENIX Security'08*, pages 31–43, 2008.

[139] Masood Masood, Arif Ghafoor, and Aditya Mathur. Conformance testing of temporal role-based access control systems. *IEEE Transactions on Dependable and Secure Computing*, 7(2):144–158, 2010.

[140] Johannes Mayer and Ralph Guderlei. On random testing of image processing applications. In *QSIC'06*, pages 85–92, 2006.

[141] Leland McInnes, John Healy, and Steve Astels. hdbscan: Hierarchical density based clustering. *Journal of Open Source Software*, 2(11):205, 2017.

[142] Sérgio Mergen. Extending the bag distance for string similarity search. *SN Comput. Sci.*, 4(2), dec 2022.

[143] Ali Mesbah, Engin Bozdag, and Arie Van Deursen. Crawling ajax by inferring user interface state changes. In *ICWE'08*, pages 122–134, 2008.

[144] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3:1–3:30, 2012.

[145] Ali Mesbah, Arie Van Deursen, and Stefan Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web*, 6(1):3, 2012.

[146] Matteo Meucci and Andrew Muller. OWASP Testing Guide v4. `https://www.owasp.org/images/1/19/OTGv4.pdf`, 2018.

[147] Microsoft Corp. Silverlight plug-ins and development tools. `https://www.microsoft.com/silverlight/`, 2019.

[148] Breno Miranda and Antonia Bertolino. Scope-aided test prioritization, selection and minimization for software reuse. *Journal of Systems and Software*, 131:528–549, 2017.

[149] MITRE. Cve-2018-1000406, concerns cwe-276, November 2018.

[150] MITRE. Cve-2018-1000409, concerns otg-sess-003, 2018.

[151] MITRE. Cve-2018-11327, concerns cwe-200, 2018.

[152] MITRE. Cve-2018-17857, concerns cwe-200, 2018.

[153] MITRE. Cve-2018-1999003, concerns otg-authz-002, November 2018.

[154] MITRE. Cve-2018-1999004, concerns otg-authz-002, November 2018.

[155] MITRE. Cve-2018-1999006, concerns cwe-138, 2018.

[156] MITRE. Cve-2018-1999046, concerns otg-authz-002, November 2018.

[157] MITRE. Cve-2020-2162, concerns otg-inpval-003, 2020.

[158] MITRE. CVE-2018-1000406, concerns OTG-AUTHN-001. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1000406, 2022.

[159] MITRE. CVE-2018-1000409, concerns OTG-SESS-003. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1000409, 2022.

[160] MITRE. CVE-2018-11327. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-11327, 2022.

[161] MITRE. CVE-2018-17857. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-17857, 2022.

[162] MITRE. CVE-2018-1999003, concerns OTG-AUTHZ-002. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999003, 2022.

[163] MITRE. CVE-2018-1999004, concerns OTG-AUTHZ-002. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999004, 2022.

[164] MITRE. CVE-2018-1999006, concerns OTG-AUTHZ-002. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999006, 2022.

[165] MITRE. CVE-2018-1999045, concerns OTG-AUTHZ-002. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999045, 2022.

[166] MITRE. CVE-2018-1999046, concerns OTG-AUTHZ-002. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999046, 2022.

[167] MITRE. CVE-2018-1999047, concerns OTG-AUTHZ-002. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999047, 2022.

[168] MITRE. CVE-2020-2162, concerns OTG-INPVAL-003. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-2162, 2022.

[169] MITRE. CWE-262. https://cwe.mitre.org/data/definitions/262.html, 2022.

[170] MITRE. CWE-521. https://cwe.mitre.org/data/definitions/521.html, 2022.

[171] Tejeddine Mouelhi, Franck Fleurey, Benoit Baudry, and Yves Le Traon. A model-based framework for security policy specification, deployment and testing. In *MODELS'08*, pages 537–552, 2008.

[172] Tejeddine Mouelhi, Yves Le Traon, and Benoit Baudry. Transforming and selecting functional test cases for security policy testing. In *ICST'09*, pages 171–180, 2009.

[173] C. Murphy, K. Shen, and G. Kaiser. Using JML Runtime Assertion Checking to Automate Metamorphic Testing in Applications without Test Oracles. In *ICST'09*, pages 436–445, 2009.

[174] Christian Murphy, Gail E Kaiser, and Lifeng Hu. Properties of machine learning applications for use in metamorphic testing. Technical report, Columbia University, 2008.

[175] Christian Murphy, Gail E Kaiser, and Lifeng Hu. Properties of machine learning applications for use in metamorphic testing. Technical report, Columbia University, 2008.

[176] Christian Murphy, Mohammad S Raunak, Andrew King, Sanjian Chen, Christopher Imbriano, Gail Kaiser, Insup Lee, Oleg Sokolsky, Lori Clarke, and Leon Osterweil. On effective testing of health care simulation software. In *SEHC'11*, pages 40–47, 2011.

[177] Mehwish Naz, Zaeem Anwaar, and Wasi Haider Butt. Automated white box test case generation for statement coverage using u-nsga-iii. In *2023 17th International Conference on Open Source Systems and Technologies (ICOSST)*, pages 1–6, 2023.

[178] Raphael Noemmer and Roman Haas. An evaluation of test suite minimization techniques. In Dietmar Winkler, Stefan Biffl, Daniel Mendez, and Johannes Bergsmann, editors, *Software Quality: Quality Intelligence in Software and Systems Engineering*, pages 51–66, Cham, 2020. Springer International Publishing.

[179] Saahil Ognawala, Martín Ochoa, Alexander Pretschner, and Tobias Limmer. MACKE: Compositional analysis of low-level vulnerabilities with symbolic execution. In *ASE'16*, pages 780–785, 2016.

[180] Oracle corp. MYSQL RDBMS engine, 2022. https://www.mysql.com/.

[181] OWASP. OTG-AUTHN-002: Testing for default credentials. `https://www.owasp.org/index.php/Testing_for_default_credentials_(OTG-AUTHN-002).`, 2022.

[182] OWASP. OTG-BUSLOGIC-006: Testing for the circumvention of work-flows. `https://www.owasp.org/index.php/Testing_for_the_Circumven\protect\discretionary{\char\hyphenchar\font}{}{}tion_of_Work_Flows_(OTG-BUSLOGIC-006).`, 2022.

[183] OWASP. OTG-INFO-010: Mapping application architecture. `https://www.owasp.org/index.php/Map_Application_Architecture_(OTG-INFO-010).`, 2022.

[184] OWASP. OTG-INPVAL-014: Testing for Buffer Overflow. https://www.owasp.org/index.php/Testing_for_Buffer_Overflow_(OTG-INPVAL-014)., 2022.

[185] OWASP. OWASP Zed Attack Proxy. https://www.zaproxy.org, 2022.

[186] OWASP. WSTG-ATHZ-01: Testing Directory Traversal File Include. https://bit.ly/3l1sRTl, 2022.

[187] OWASP. WSTG-ATHZ-02: Testing for Bypassing Authorization Schema. https://bit.ly/2Y6SloC, 2022.

[188] OWASP. WSTG-SESS-03: Testing for Session Fixation. https://bit.ly/34myBkN, 2022.

[189] Rongqi Pan, Taher A. Ghaleb, and Lionel Briand. Atm: Black-box test case minimization based on test code similarity and evolutionary search. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 1700–1711. IEEE Press, 2023.

[190] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, Graz, Austria, April 2015. IEEE.

[191] Mauro Pezze and Cheng Zhang. Automated test oracles: A survey. *Advances in Computers*, 95:1–48, 2014.

[192] Beatriz Pontes, Raúl Giráldez, and Jesús S. Aguilar-Ruiz. Biclustering on expression data: A review. *Journal of Biomedical Informatics*, 57:163–180, 2015.

[193] Portswigger. Burp suite. https://portswigger.net/burp, 2022.

[194] Portswigger. Burp suite scanning (crawling) feature. https://portswigger.net/burp/documentation/desktop/scanning, 2022.

[195] Portswigger. 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, 2022.

[196] Laura L Pullum and Ozgur Ozmen. Early results from metamorphic testing of epidemiological models. In *BioMedCom'12*, pages 62–67, 2012.

[197] Sriram Raghavan and Hector Garcia-Molina. Crawling the hidden web. In *VLDB'01*, pages 129–138, 2000.

[198] Paul Ralph and Ewan Tempero. Construct Validity in Software Engineering Research and Software Metrics. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, volume Part F1377, pages 13–23, New York, NY, USA, jun 2018. ACM.

[199] Arvind Ramanathan, Chad A Steed, and Laura L Pullum. Verification of compartmental epidemiological models using metamorphic testing, model checking and visual analytics. In *BioMedCom'12*, pages 68–73, 2012.

[200] Guy Rosen. Facebook Security Update on 'View As' Vulnerability. `https://newsroom.fb.com/news/2018/09/security-update/`, 2019.

[201] Prashanta Saha and Upulee Kanewala. Fault detection effectiveness of source test case generation strategies for metamorphic testing. In *Proceedings of the 3rd International Workshop on Metamorphic Testing*, MET '18, page 2–9, New York, NY, USA, 2018. Association for Computing Machinery.

[202] MIP Salas and Eliane Martins. Security testing methodology for vulnerabilities detection of XSS in web services and WS-security. *ENTCS*, pages 133–154, 2014.

[203] Joanna CS Santos, Anthony Peruma, Mehdi Mirakhorli, Matthias Galstery, Jairo Veloz Vidal, and Adriana Sejfia. Understanding software vulnerabilities related to architectural security tactics: An empirical investigation of chromium, php and thunderbird. In *ICSA'17*, pages 69–78, 2017.

[204] Joanna CS Santos, Katy Tarrit, and Mehdi Mirakhorli. A catalog of security architecture weaknesses. In *ICSAW'17*, pages 220–223, 2017.

[205] Sergio Segura, Juan C. Alonso, Alberto Martin-Lopez, Amador Durán, Javier Troya, and Antonio Ruiz-Cortés. Automated generation of metamorphic relations for query-based systems. In *Proceedings of the 7th International Workshop on Metamorphic Testing*, MET '22, page 48–55, New York, NY, USA, 2023. Association for Computing Machinery.

[206] Sergio Segura, Amador Durán, Ana B Sánchez, Daniel Le Berre, Emmanuel Lonca, and Antonio Ruiz-Cortés. Automated metamorphic testing of variability analysis tools. *Software Testing, Verification and Reliability*, 25(2):138–163, 2015.

[207] Sergio Segura, Amador Durán, Javier Troya, and Antonio Ruiz Cortés. A template-based approach to describing metamorphic relations. In *MET'17*, pages 3–9, 2017.

[208] Sergio Segura, Amador Durán, Javier Troya, and Antonio Ruiz-Cortés. Metamorphic relation template v1. 0. *Applied Software Engineering Research Group, Tech. Rep. ISA-17-TR-01*, 2017.

[209] Sergio Segura, Amador Durán, Javier Troya, and Antonio Ruiz-Cortés. Metamorphic relation patterns for query-based systems. In *2019 IEEE/ACM 4th International Workshop on Metamorphic Testing (MET)*, pages 24–31, 2019.

[210] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824, Sep. 2016.

[211] Sergio Segura, Robert M Hierons, David Benavides, and Antonio Ruiz-Cortés. Automated test data generation on the analyses of feature models: A metamorphic testing approach. In *ICST'10*, pages 35–44, 2010.

[212] Sergio Segura, Robert M Hierons, David Benavides, and Antonio Ruiz-Cortés. Automated metamorphic testing on the analyses of feature models. *Information and Software Technology*, 53(3):245–258, 2011.

[213] Sergio Segura, José A Parejo, Javier Troya, and Antonio Ruiz-Cortés. Metamorphic testing of RESTful web APIs. *IEEE Transactions on Software Engineering*, 44(11):1083–1099, 2017.

[214] Selenium Web Testing Framework, `https://www.seleniumhq.org/`, 2018.

[215] KY Sim, WKS Pao, and C Lin. Metamorphic testing using geometric interrogation technique and its application. *ECTI Transactions on Computer and Information Technology (ECTI-CIT)*, 1(2):91–95, 2005.

[216] Petr Slavík. A tight analysis of the greedy algorithm for set cover. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 435–441, New York, NY, USA, 1996. Association for Computing Machinery.

[217] SonarSource. Sonarqube: Code Quality and Code Security toolset. `https://www.sonarqube.org/`, 2022.

[218] Matt Staats, Michael W Whalen, and Mats PE Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In *ICSE'11*, pages 391–400, 2011.

[219] Chang-Ai Sun, Hepeng Dai, Huai Liu, and Tsong Yueh Chen. Feedback-directed metamorphic testing. *ACM Trans. Softw. Eng. Methodol.*, 32(1), Feb 2023.

[220] Chang-Ai Sun, An Fu, Pak-Lok Poon, Xiaoyuan Xie, Huai Liu, and Tsong Yueh Chen. Metric$^+$+: A metamorphic relation identification technique based on input plus output domains. *IEEE Transactions on Software Engineering*, 47(9):1764–1785, 2021.

[221] Chang-ai Sun, Baoli Liu, An Fu, Yiqiang Liu, and Huai Liu. Path-directed source test case generation and prioritization in metamorphic testing. *Journal of Systems and Software*, 183:111091, 2022.

[222] Chang-ai Sun, Guan Wang, Baohong Mu, Huai Liu, ZhaoShun Wang, and Tsong Yueh Chen. Metamorphic testing for web services: Framework and a case study. In *ICWS'11*, pages 283–290, 2011.

[223] Chang-ai Sun, Zuoyi Wang, and Guan Wang. A property-based testing framework for encryption programs. *Frontiers of Computer Science*, 8(3):478–489, 2014.

[224] Synopsys Inc. Description of the openssl heartbleed vulnerability. `http://heartbleed.com/`, 2022.

[225] Ari Takanen, Jared D Demott, Charles Miller, and Atte Kettunen. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2018.

[226] Joshua Y.S. Tang, Andrian Yang, Tsong Yueh Chen, and Joshua W.K. Ho. Harnessing multiple source test cases in metamorphic testing: A case study in bioinformatics. In *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*, pages 10–13, Buenos Aires, Argentina, 2017. IEEE.

[227] Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. An automatic testing approach for compiler based on metamorphic testing technique. In *APSEC'10*, pages 270–279, 2010.

[228] Julian Thomé, Lwin Khin Shar, Domenico Bianculli, and Lionel Briand. Search-driven string constraint solving for vulnerability detection. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 198–208, Buenos Aires, Argentina, 2017. IEEE.

[229] Gu Tian-yang, Shi Yin-Sheng, and Fang You-yuan. Research on software security testing. *World Academy of Science, Engineering and Technology*, 70(69):647–651, 2010.

[230] Omer Tripp, Omri Weisman, and Lotem Guy. Finding your way in the testing jungle: A learning approach to web security testing. In *ISSTA'13*, pages 347–357, 2013.

[231] TH Tse and Stephen S Yau. Testing context-sensitive middleware-based software applications. In *COMPSAC'04*, pages 458–466, 2004.

[232] András Vargha and Harold D. Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[233] Vijay V. Vazirani. *Approximation algorithms*. Springer-Verlag, Berlin, Heidelberg, 2001.

[234] Jeffrey M. Voas and Keith W Miller. Software testability: The new verification. *IEEE software*, 12(3):17–28, 1995.

[235] James Walker. Teen hacker scoops $4,500 bug bounty for Facebook flaw that allowed attackers to unmask page admins, 2021. https://portswigger.net/daily-swig/teen-hacker-scoops-4-500-bug-bounty-for-facebook-flaw-that-allowed-attackers-to-unmask-page-admins.

[236] Shuai Wang, Shaukat Ali, and Arnaud Gotlieb. Cost-effective test suite minimization in product lines using search techniques. *Journal of Systems and Software*, 103:370–391, 2015.

[237] Jon Whittle, Duminda Wijesekera, and Mark Hartong. Executable misuse cases for modeling security concerns. In *ICSE'08*, pages 121–130, 2008.

[238] Guido Wimmel and Jan Jürjens. Specification-based test generation for security-critical systems using mutations. In *ICFEM'02*, pages 471–482, 2002.

[239] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, Heidelberg, Germany, 2012.

[240] Robin Wood. Damn vulnerable web application (dvwa), 2022. https://dvwa.co.uk/.

[241] Emma Woollacott. Facebook account takeover: Researcher scoops 40k bug bounty for chained exploit, 2022. https://portswigger.net/daily-swig/facebook-account-takeover-researcher-scoops-40k-bug-bounty-for-chained-exploit.

[242] Xiaoyuan Xie, Joshua Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. Application of metamorphic testing to supervised classifiers. In *QSIC'09*, pages 135–144, 2009.

[243] Dianxiang Xu and Kendall E Nygard. Threat-driven modeling and verification of secure software using aspect-oriented petri nets. *IEEE Transactions on Software Engineering*, 32(4):265–278, 2006.

[244] Dianxiang Xu, Lijo Thomas, Michael Kent, Tejeddine Mouelhi, and Yves Le Traon. A model-based approach to automated testing of access control policies. In *SACMAT'12*, pages 209–218, 2012.

[245] Dianxiang Xu, Manghui Tu, Michael Sanford, Lijo Thomas, Daniel Woodraska, and Weifeng Xu. Automated security test generation with formal threat models. *IEEE Transactions on Dependable and Secure Computing*, 9(4):526–540, 2012.

[246] Dianxiang Xu, Weifeng Xu, Michael Kent, Lijo Thomas, and Linzhang Wang. An automated test generation technique for software quality assurance. *IEEE Transactions on Reliability*, 64(1):247–268, 2015.

[247] Liming Xu, Dave Towey, Andrew P French, Steve Benford, Zhi Quan Zhou, and Tsong Yueh Chen. Using metamorphic relations to verify and enhance artcode classification. *Journal of Systems and Software*, 182:111060, 2021.

[248] Yi Yao, Song Huang, and Mengyu Ji. Research on metamorphic testing for oracle problem of integer bugs. In *AISC'12*, pages 95–100. 2012.

[249] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, mar 2012.

[250] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. The fuzzing book. In *The Fuzzing Book*. Saarland University, 2019. Retrieved 2019-09-09 16:42:54+02:00.

[251] Bo Zhang, Hongyu Zhang, Junjie Chen, Dan Hao, and Pablo Moscato. Automatic discovery and cleansing of numerical metamorphic relations. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 235–245, Cleveland, USA, 2019. IEEE.

[252] Man Zhang, Shaukat Ali, and Tao Yue. Uncertainty-wise test case generation and minimization for cyber-physical systems. *Journal of Systems and Software*, 153:1–21, 2019.

[253] Miao Zhang, Jacky Wai Keung, Tsong Yueh Chen, and Yan Xiao. Validating class integration test order generation systems with metamorphic testing. *Information and Software Technology*, 132:106507, 2021.

[254] Shengnan Zhang, Yan Hu, and Guangrong Bian. Research on string similarity algorithm based on levenshtein distance. In *2017 IEEE 2nd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, pages 2247–2251, Chongqing, China, 2017. IEEE.

[255] Jinyi Zhou, Kun Qiu, Zheng Zheng, Tsong Yueh Chen, and Pak-Lok Poon. Using metamorphic testing to evaluate dnn coverage criteria. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 147–148, Coimbra, Portugal, 2020. IEEE.

[256] Zhi Quan Zhou, Liqun Sun, Tsong Yueh Chen, and Dave Towey. Metamorphic relations for enhancing system understanding and use. *IEEE Transactions on Software Engineering*, 46(10):1120–1154, 2018.

[257] Zhi Quan Zhou, ShuJia Zhang, Markus Hagenbuchner, TH Tse, Fei-Ching Kuo, and Tsong Yueh Chen. Automated functional testing of online search services. *Software: Testing, Verification and Reliability*, 22(4):221–243, 2012.

[258] H. Zhu. Jfuzz: A tool for automated java unit testing based on data mutation and metamorphic testing methods. In *TSA'15*, pages 8–15, 2015.

[259] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. Spea2: Improving the strength pareto evolutionary algorithm. Technical Report 103, Computer Engineering and Communication Networks Lab (TIK), Swiss Federal Institute of Technology (ETH), Zurich, 2001.

Appendix

We prove in this appendix theoretical results mentioned in this Chapter and which demonstrates the correctness of the AIM approach.

A.1 Input Coverage and Cost

Lemma 1 (Non-Empty Coverage)**.** *For every input in, we have* $Cover(in) \neq \varnothing$.

Proof. We consider only inputs containing at least one action (Section 3.6.2). Let $act_1 = action(in, 1)$ be the first action of *in*. act_1 has an output in $outCl_1 = OutputClass(in, 1)$ (Section 3.6.2).act_1 is in the action set $actSet_1 = ActionSet(outCl_1)$ (Section 3.6.3). After clustering of this action set, let $bl_1 = Subclass(in, 1) = ActionSubclass(act_1, actSet_1)$ be the input block of the action act_1 executed in *in* (Section 3.6.3). Therefore, we have $bl_1 \in Cover(in)$. □

Lemma 2 (Cost is Positive)**.** *For every input in, we have* $cost(in) > 0$. *For every input set* I*, we have* $cost(I) \geq 0$*, and* $cost(I) = 0$ *if and only if* $I = \varnothing$.

Proof. Since we removed inputs with no cost (Section 3.3.1), for each input *in*, $cost(in) > 0$. The cost of an input set is the sum of the cost of its inputs, hence $cost(I) \geq 0$. In particular, if $I = \varnothing$, then $cost(I) = 0$. Finally, because the cost is positive, the only case where $cost(I) = 0$ is when $I = \varnothing$. □

A.2 Redundancy

In this section, we prove that our characterization of redundancy is sound regarding the coverage of an input set (Proposition 1), then we introduce Lemma 4 which is useful to prove Lemma 6 in Section A.3.

First, if an input *in* is in the considered input set I, then its redundancy is not negative. Indeed, there is always at least one input in I that covers the input blocks covered by *in*, which is *in* itself.

Lemma 3 (Redundancy is Non-negative)**.** *Let* I *be an input set and in be an input. If* $in \in I$*, then* $redundancy(in, I) \geq 0$.

Proof. Let *in* be an input in I. Let $bl \in Cover(in)$ be any block covered by *in*. Because $bl \in Cover(in)$, we have that $in \in Inputs(bl)$ (Section 3.3.1). Moreover, $in \in I$, so $in \in Inputs(bl) \cap I$, thus we have $superpos(bl, I) \geq 1$ (Section 3.3.3). Therefore, for any $bl \in Cover(in)$ we have $superpos(bl, I) \geq 1$. So, $\min\{superpos(bl, I) \mid bl \in Cover(in)\} \geq 1$. By subtracting 1, we have $redundancy(in, I) \geq 0$ (Section 3.3.3). □

We prove that our characterization of redundancy is sound regarding the coverage of an input set. In other words, if an input is redundant in an input set, then it can be removed without reducing the coverage of the input set:

Proposition 1 (Redundancy Soundness)**.** *Let $I \subseteq I_{init}$ be an input set and $in \in I$. If $in \in Redundant(I)$, then $Cover(I \setminus \{in\}) = Cover(I)$.*

Proof. $Cover(I) = Cover(I \setminus \{in\}) \cup Cover(in)$ (Section 3.6.3). We assume that in is redundant in I. So, according to Lemma 3, we have $redundancy(in, I) > 0$ (Section 3.3.3). Thus, for each $bl \in Cover(in)$, we have that $superpos(bl, I) \geq 2$. Hence, according to our definition of superposition (Section 3.3.3), there are at least two inputs in_1 and in_2 in I which also belong to $Inputs(bl)$. Because we have $bl \in Cover(in)$, we have that $in \in Inputs(bl)$ (Section 3.3.1). So, in is one of these inputs. We assume this is the first one i.e., $in_1 = in$. Therefore, for each $bl \in Cover(in)$, there exists an input $in_2 \in Inputs(bl) \cap I$ which is not in. We denote $in_2 = in_{bl}$ this input. For each $bl \in Cover(in)$, we have $in_{bl} \in Inputs(bl)$. So, we have $bl \in Cover(in_{bl})$ (Section 3.3.1). Moreover, $in_{bl} \in I$ and is not in, so is in $I \setminus \{in\}$. Thus (Section 3.6.3) we have $bl \in Cover(I \setminus \{in\})$. Therefore, $Cover(in) \subseteq Cover(I \setminus \{in\})$. Finally, $Cover(I) = Cover(I \setminus \{in\}) \cup Cover(in) = Cover(I \setminus \{in\})$. □

Finally, removing an input in an input set may update the redundancy of other inputs:

Lemma 4 (Removing an Input)**.** *Let I be an input set and $in_2, in_1 \in I$ be two inputs in this input set.*

$$\begin{aligned} redundancy(in_1, I) - 1 &\leq redundancy(in_1, I \setminus \{in_2\}) \\ &\leq redundancy(in_1, I) \end{aligned}$$

Proof. Let $bl \in Cover(in_1)$. We denote $c_1 = |Inputs(bl) \cap I|$ and $c_2 = |Inputs(bl) \cap (I \setminus \{in_2\})|$. If $in_2 \in Inputs(bl)$ then $c_2 = c_1 - 1$. Otherwise, $c_2 = c_1$. Therefore, for every $bl \in Cover(in_1)$ we have $c_1 - 1 \leq c_2 \leq c_1$. This includes the input blocks minimizing the cardinality in the definition of redundancy (Section 3.3.3). Hence the result. □

A.3 Valid Orders of Removal Steps

In this section, we first prove Lemma 5 that establishes that orders of removal steps contain inputs without repetition. Then, we introduce subsequences and prove Lemma 7, used in the rest of the section. Finally, we prove Theorem 1 presented in Section 3.3.3.

Lemma 5 (Order without Repetition)**.** *Let I be an input set. If $[in_1, \ldots, in_n] \in ValidOrders(I)$, then $in_1, \ldots, in_n$ are distinct.*

Proof. The proof is done by induction on n.

If $n = 0$, then $[in_1, \ldots, in_n] = [\,]$ is empty, hence there are no two identical inputs.

We now consider $[in_1, \ldots, in_n, in_{n+1}] \in ValidOrders(I)$. By induction hypothesis, $in_1, \ldots, in_n$ are distinct. According to the definition of valid order of removal steps (Section 3.3.3), we have $[in_1, \ldots, in_n, in_{n+1}]$ only if $in_{n+1} \in Redundant(I \setminus \{in_1, \ldots, in_n\})$. Since $Redundant(I) = \{in \in I \mid redundancy(in, I) > 0\}$, we have $in_{n+1} \in I \setminus \{in_1, \ldots, in_n\}$.

Therefore, in_{n+1} is distinct from the previous inputs $in_1, \ldots, in_n$. Hence the result, which concludes the induction step. □

Lemma 6 (Redundant Inputs After Reduction)**.** *Let I be an input set. For each subset of inputs $\{in_1, \ldots, in_n\} \subseteq I$, we have $Redundant(I \setminus \{in_1, \ldots, in_n\}) \subseteq Redundant(I)$.*

Proof. The proof is done by induction on n.

If $n = 0$ then $I \setminus \{in_1, \ldots, in_n\} = I$, hence the result.

Otherwise, we assume by induction that $Redundant(I \setminus \{in_1, \ldots, in_n\}) \subseteq Redundant(I)$.

According to Lemma 4, redundancies can only decrease when performing a removal step. So, for every input $in_0 \in Redundant(I \setminus \{in_1, \ldots, in_n, in_{n+1}\})$, we have:

$$\begin{aligned} 0 &< redundancy(in_0, I \setminus \{in_1, \ldots, in_n, in_{n+1}\}) \\ &\leq redundancy(in_0, I \setminus \{in_1, \ldots, in_n\}) \end{aligned}$$

So, $Redundant(I \setminus \{in_1, \ldots, in_n, in_{n+1}\}) \subseteq Redundant(I \setminus \{in_1, \ldots, in_n\}) \subseteq Redundant(I)$. □

Since orders of removal steps contain inputs without repetition (Lemma 5), the index $index(in, \ell)$ of an input in in a removal order ℓ containing in is well defined. We leverage this to define subsequences.

Definition 1 (Subsequences)**.** Let $[in_1, \ldots, in_n]$ and $[in'_1, \ldots, in'_m]$ be two orders of removal steps. We say that $[in'_1, \ldots, in'_m]$ is a *subsequence* of $[in_1, \ldots, in_n]$ if $[in'_1, \ldots, in'_m]$ can be obtained by removing some elements of $[in_1, \ldots, in_n]$, i.e., if $\{in \in [in'_1, \ldots, in'_m]\} \subseteq \{in \in [in_1, \ldots, in_n]\}$ and, for any two inputs $in_i, in_j \in [in'_1, \ldots, in'_m]$, if $index(in_i, [in_1, \ldots, in_n]) \leq index(in_j, [in_1, \ldots, in_n])$, then $index(in_i, [in'_1, \ldots, in'_m]) \leq index(in_j, [in'_1, \ldots, in'_m])$, where $index(in, \ell)$ is the index of input in in ℓ.

Lemma 7. *Let I be an input set and $[in_1, \ldots, in_n], [in'_1, \ldots, in'_m]$ be two orders of removal steps in I. If $[in_1, \ldots, in_n] \in ValidOrders(I)$ and $[in'_1, \ldots, in'_m]$ is a subsequence of $[in_1, \ldots, in_n]$, then $[in'_1, \ldots, in'_m] \in ValidOrders(I)$.*

Proof. The proof is done by induction on m.

If $m = 0$ then $[in'_1, \ldots, in'_m] = [\,] \in ValidOrders(I)$.

Otherwise, we consider $[in'_1, \ldots, in'_m, in'_{m+1}]$ and we assume by induction that $[in'_1, \ldots, in'_m] \in ValidOrders(I)$.

Because $[in'_1, \ldots, in'_m, in'_{m+1}]$ is a subsequence of $[in_1, \ldots, in_n]$, there exists an index i such that $in_i = in'_{m+1}$. By definition of valid removal steps (Section 3.3.3), we have $in_i \in Redundant(I \setminus \{in_1, \ldots, in_{i-1}\})$.

Moreover, $[in'_1, \ldots, in'_m, in'_{m+1}]$ is a subsequence of $[in_1, \ldots, in_i]$, so we have $\{in \in [in'_1, \ldots, in'_m]\} \subseteq \{in \in [in_1, \ldots, in_{i-1}]\}$. Hence, according to Lemma 6 we have $Redundant(I \setminus \{in_1, \ldots, in_{i-1}\}) \subseteq Redundant(I \setminus \{in'_1, \ldots, in'_m\})$. Therefore, $in'_{m+1} = in_i \in Redundant(I \setminus \{in'_1, \ldots, in'_m\})$.

$[in'_1, \ldots, in'_m] \in ValidOrders(I)$ and $in'_{m+1} \in Redundant(I \setminus \{in'_1, \ldots, in'_m\})$, so, according to our definition for valid orders of removal steps (Section 3.3.3), we have $[in'_1, \ldots, in'_m, in'_{m+1}] \in ValidOrders(I)$. □

Finally, we prove in Proposition 2 that inputs in a valid order of removal steps can be rearranged in any different order and the rearranged order of removal steps is also valid.

Lemma 8 (Transposition of a Valid Order)**.** *Let I be an input set.*

$$\begin{aligned} &\textit{If } [in_1, \ldots, in_i, \ldots, in_j, \ldots, in_n] \in \textit{ValidOrders}(I) \\ &\textit{and } 1 \leq i < j \leq n, \\ &\textit{then } [in_1, \ldots, in_j, \ldots, in_i, \ldots, in_n] \in \textit{ValidOrders}(I) \end{aligned}$$

where in_i and in_j were exchanged and the other inputs are left unchanged.

Proof. The proof is done in four steps.

1) $[in_1, \ldots, in_{i-1}, in_j]$ is a subsequence of $[in_1, \ldots, in_{i-1}, in_i, \ldots, in_j, \ldots, in_n] \in \textit{ValidOrders}(I)$. So, according to Lemma 7, we have $[in_1, \ldots, in_{i-1}, in_j] \in \textit{ValidOrders}(I)$. Note that the first step even applies to the case $i = 1$.

2) If $j = i+1$, then one can go directly to the third step. Otherwise, for each $i < k < j$, we denote:

$$\begin{aligned} I_{i+1} &\stackrel{\text{def}}{=} I \setminus \{in_1, \ldots, in_{i-1}\} \\ I_{k+1} &\stackrel{\text{def}}{=} I_k \setminus \{in_k\} \end{aligned}$$

and, for the sake of conciseness, $I_k^i = I_k \setminus \{in_i\}$ and $I_k^j = I_k \setminus \{in_j\}$.

We now prove by induction on $i \leq k \leq j-1$ that:

$$[in_1, \ldots, in_{i-1}, in_j, in_{i+1}, \ldots, in_k] \in \textit{ValidOrders}(I)$$

For the initialization $k = i$, we have from the first step:

$$[in_1, \ldots, in_{i-1}, in_j] \in \textit{ValidOrders}(I)$$

We now assume by induction hypothesis on $i \leq k < j-1$ that:

$$[in_1, \ldots, in_{i-1}, in_j, in_{i+1}, \ldots, in_k] \in \textit{ValidOrders}(I)$$

We first prove, for each $bl \in \textit{Cover}(in_{k+1})$, that:

$$\left|\textit{Inputs}(bl) \cap I_{k+1}^j\right| > 1$$

According to Lemma 5, $in_1, \ldots, in_i, \ldots, in_j, \ldots, in_n$ are distinct. We consider two cases.

a) We assume $bl \in \textit{Cover}(in_j)$.

Because $I_j^i = I \setminus \{in_1, \ldots, in_{i-1}, in_i, in_{i+1}, \ldots, in_{j-1}\}$
and $I_{k+1}^j = I \setminus \{in_1, \ldots, in_{i-1}, in_{i+1}, \ldots, in_k, in_j\}$, we have $I_j^i \cup \{in_{k+1}\} \subseteq I_{k+1}^j \cup \{in_j\}$. Thus, for each $bl \in \textit{Cover}(in_j) \cap \textit{Cover}(in_{k+1})$:

$$\left|\textit{Inputs}(bl) \cap I_j^i\right| + 1 \leq \left|\textit{Inputs}(bl) \cap I_{k+1}^j\right| + 1$$

Because $[in_1, \ldots, in_i, \ldots, in_j, \ldots, in_n] \in \mathit{ValidOrders}(I)$, we have:

$$\mathit{redundancy}(in_j, I_j^i) > 0$$

So, for each $bl \in \mathit{Cover}(in_j)$, we have:

$$\left|\mathit{Inputs}(bl) \cap I_j^i\right| > 1$$

Thus, for each $bl \in \mathit{Cover}(in_j) \cap \mathit{Cover}(in_{k+1})$, we have:

$$\left|\mathit{Inputs}(bl) \cap I_{k+1}^j\right| > 1$$

b) We assume $bl \notin \mathit{Cover}(in_j)$.

For each $bl \in \mathit{Cover}(in_{k+1}) \setminus (\mathit{Cover}(in_i) \cup \mathit{Cover}(in_j))$, we have:

$$\left|\mathit{Inputs}(bl) \cap I_{k+1}^i\right| = \left|\mathit{Inputs}(bl) \cap I_{k+1}^j\right|$$

Moreover, for each $bl \in \mathit{Cover}(in_i) \setminus \mathit{Cover}(in_j)$, we have:

$$\left|\mathit{Inputs}(bl) \cap I_{k+1}^i\right| + 1 = \left|\mathit{Inputs}(bl) \cap I_{k+1}^j\right|$$

So, for each $bl \in \mathit{Cover}(in_{k+1}) \setminus \mathit{Cover}(in_j)$, we have:

$$\left|\mathit{Inputs}(bl) \cap I_{k+1}^i\right| \leq \left|\mathit{Inputs}(bl) \cap I_{k+1}^j\right|$$

Because $[in_1, \ldots, in_i, \ldots, in_j, \ldots, in_n] \in \mathit{ValidOrders}(I)$, we have:

$$\mathit{redundancy}(in_{k+1}, I_{k+1}^i) > 0$$

So, for each $bl \in \mathit{Cover}(in_{k+1})$, we have:

$$\left|\mathit{Inputs}(bl) \cap I_{k+1}^i\right| > 1$$

Thus, for each $bl \in \mathit{Cover}(in_{k+1}) \setminus \mathit{Cover}(in_j)$, we have:

$$\left|\mathit{Inputs}(bl) \cap I_{k+1}^j\right| > 1$$

So, we proved by case, for each $bl \in \mathit{Cover}(in_{k+1})$, that:

$$\left|\mathit{Inputs}(bl) \cap I_{k+1}^j\right| > 1$$

Thus, we have $\mathit{redundancy}(in_{k+1}, I_{k+1}^j) > 0$.

Moreover, according to Lemma 5, $in_1, \ldots, in_i, \ldots, in_j, \ldots, in_n$ are distinct, so $in_{k+1} \in I_{k+1}^j$. Hence, $in_{k+1} \in \mathit{Redundant}(I_{k+1}^j)$. By induction hypothesis, we have:

$$[in_1, \ldots, in_{i-1}, in_j, in_{i+1}, \ldots, in_k] \in \mathit{ValidOrders}(I)$$

So, according to the definition of valid removal steps Section 3.3.3:

$$[in_1, \ldots, in_{i-1}, in_j, in_{i+1}, \ldots, in_k, in_{k+1}] \in \mathit{ValidOrders}(I)$$

which concludes the induction step.

Therefore, we proved by induction:

$$[in_1, \ldots, in_{i-1}, in_j, in_{i+1}, \ldots, in_{j-1}] \in \mathit{ValidOrders}(I)$$

3) We now prove that, for each $bl \in \mathit{Cover}(in_i)$, we have:

$$\left|\mathit{Inputs}(bl) \cap I_j^j\right| > 1$$

where I_j^i and I_j^j are defined in step two. We consider two cases.

a) We assume $bl \in \mathit{Cover}(in_j)$. Because $I_j^i \cup \{in_i\} = I_j^j \cup \{in_j\}$, for each $bl \in \mathit{Cover}(in_i) \cap \mathit{Cover}(in_j)$, we have:

$$\left|\mathit{Inputs}(bl) \cap I_j^i\right| + 1 = \left|\mathit{Inputs}(bl) \cap I_j^j\right| + 1$$

Because $[in_1, \ldots, in_i, \ldots, in_j, \ldots, in_n] \in \mathit{ValidOrders}(I)$, we have:

$$\mathit{redundancy}(in_j, I_j^i) > 0$$

So, for each $bl \in \mathit{Cover}(in_j)$, we have:

$$\left|\mathit{Inputs}(bl) \cap I_j^i\right| > 1$$

Thus, for each $bl \in \mathit{Cover}(in_i) \cap \mathit{Cover}(in_j)$, we have:

$$\left|\mathit{Inputs}(bl) \cap I_j^j\right| > 1$$

b) We assume $bl \notin \mathit{Cover}(in_j)$. In that case, for each $bl \in \mathit{Cover}(in_i) \setminus \mathit{Cover}(in_j)$, we have:

$$\left|\mathit{Inputs}(bl) \cap I_j^i\right| + 1 = \left|\mathit{Inputs}(bl) \cap I_j^j\right|$$

We consider two cases.

i) There exists $i + 1 \leq k \leq j - 1$ such that $bl \in \mathit{Cover}(in_k)$. In that case, let $k_{\max}$ be the largest. So, we have:

$$\left(\mathit{Inputs}(bl) \cap I_j^i\right) \cup \{in_{k_{\max}}\} = \mathit{Inputs}(bl) \cap I_{k_{\max}}^i$$

Hence:

$$\left|\mathit{Inputs}(bl) \cap I_j^i\right| + 1 = \left|\mathit{Inputs}(bl) \cap I_{k_{\max}}^i\right|$$

Thus:

$$\left|\mathit{Inputs}(bl) \cap I_j^j\right| = \left|\mathit{Inputs}(bl) \cap I_{k_{\max}}^i\right|$$

Because $[in_1, \ldots, in_i, \ldots, in_j, \ldots, in_n] \in \mathit{ValidOrders}(I)$, we have:

$$\mathit{redundancy}(in_{k_{\max}}, I^i_{k_{\max}}) > 0$$

So, because $bl \in \mathit{Cover}(in_{k_{\max}})$, we have:

$$\left|\mathit{Inputs}(bl) \cap I^i_{k_{\max}}\right| > 1$$

Finally:

$$\left|\mathit{Inputs}(bl) \cap I^j_j\right| > 1$$

ii) Otherwise, for each $i+1 \leq k \leq j-1$, we have $bl \notin \mathit{Cover}(in_k)$. Note that this is also the case if $j = i+1$. In that case, we have:

$$\left(\mathit{Inputs}(bl) \cap I^i_j\right) \cup \{in_i\} = \mathit{Inputs}(bl) \cap I_i$$

where $I_i = I \setminus \{in_1, \ldots, in_{i-1}\}$. So:

$$\left|\mathit{Inputs}(bl) \cap I^i_j\right| + 1 = |\mathit{Inputs}(bl) \cap I_i|$$

Thus:

$$\left|\mathit{Inputs}(bl) \cap I^j_j\right| = |\mathit{Inputs}(bl) \cap I_i|$$

Because $[in_1, \ldots, in_i, \ldots, in_j, \ldots, in_n] \in \mathit{ValidOrders}(I)$, we have:

$$\mathit{redundancy}(in_i, I_i) > 0$$

So, because $bl \in \mathit{Cover}(in_i)$, we have:

$$|\mathit{Inputs}(bl) \cap I_i| > 1$$

Finally:

$$\left|\mathit{Inputs}(bl) \cap I^j_j\right| > 1$$

This concludes the proof by case for $bl \in \mathit{Cover}(in_i) \setminus \mathit{Cover}(in_j)$. Therefore, we proved by case that, for each $bl \in \mathit{Cover}(in_i)$, we have:

$$\left|\mathit{Inputs}(bl) \cap I^j_j\right| > 1$$

Thus, we have $\mathit{redundancy}(in_i, I^j_j) > 0$.

Moreover, according to Lemma 5, $in_1, \ldots, in_i, \ldots, in_j, \ldots, in_n$ are distinct, so $in_i \in I^j_j$. Thus, $in_i \in \mathit{Redundant}(I^j_j)$.

Finally, we have from the second step (or the first one, if $j = i+1$):

$$[in_1, \ldots, in_{i-1}, in_j, in_{i+1}, \ldots, in_{j-1}] \in \mathit{ValidOrders}(I)$$

Therefore, according to the definition of valid orders of removal steps (Section 3.3.3):

$$[in_1, \ldots, in_{i-1}, in_j, in_{i+1}, \ldots, in_{j-1}, in_i] \in \mathit{ValidOrders}(I)$$

4) Finally, if $j = n$ then the proof is complete. Otherwise, we prove by induction on $j \leq k \leq n$ that:

$$[in_1, \ldots, in_j, \ldots, in_i, in_{j+1}, \ldots, in_k] \in \mathit{ValidOrders}(I)$$

For the initialization $k = j$, we have from the third step:

$$[in_1, \ldots, in_j, \ldots, in_i] \in \mathit{ValidOrders}(I)$$

We now assume by induction hypothesis on $j \leq k < n$ that:

$$[in_1, \ldots, in_j, \ldots, in_i, in_{j+1}, \ldots, in_k] \in \mathit{ValidOrders}(I)$$

Because $[in_1, \ldots, in_i, \ldots, in_j, \ldots, in_n] \in \mathit{ValidOrders}(I)$, we have:

$$in_{k+1} \in \mathit{Redundant}(I \setminus \{in_1, \ldots, in_i, \ldots, in_j, \ldots, in_k\})$$

Moreover, we have:

$$\begin{aligned} &\{in_1, \ldots, in_i, \ldots, in_j, \ldots, in_k\} \\ &= \{in_1, \ldots, in_j, \ldots, in_i, \ldots, in_k\} \end{aligned}$$

So:

$$in_{k+1} \in \mathit{Redundant}(I \setminus \{in_1, \ldots, in_j, \ldots, in_i, \ldots, in_k\})$$

Thus, according to the definition of valid orders of removal steps (Section 3.3.3):

$$\begin{aligned} &[in_1, \ldots, in_j, \ldots, in_i, in_{j+1}, \ldots, in_k, in_{k+1}] \\ &\in \mathit{ValidOrders}(I) \end{aligned}$$

which concludes the induction step.

Therefore, we proved by induction the lemma:

$$[in_1, \ldots, in_j, \ldots, in_i, in_{j+1}, \ldots, in_n] \in \mathit{ValidOrders}(I)$$

□

The issue with our definition of the gain (Section 3.3.3) is that, to compute $\mathit{gain}(I)$ of an input set I, one has to try all the possible order of removal steps to determine which ones are valid. If I contains n inputs and we consider orders of $0 \leq k \leq n$ removal steps, there are $n \times \cdots \times (n - k + 1) = \frac{!n}{!(n-k)}$ possibilities to investigate, which is usually large.

Thus, we present an optimization to reduce the cost of computing the gain. First, we prove in Proposition 2 that, while the order of inputs matters to determine if an order of removal steps is valid, it does not matter anymore once we know the order is valid. In other words, inputs in a valid order of removal steps may be rearranged in any different order, and the rearranged order of removal steps would be valid. Formally, if $[in_1, \ldots, in_n]$ is a valid order of removal steps, then $[in_{\sigma(1)}, \ldots, in_{\sigma(n)}]$ is also a valid order, when σ denotes a permutation of the n inputs, i.e., the same inputs but (potentially) in a different order:

Proposition 2 (Permutation of a Valid Order)**.** *Let I be an input set. If $[in_1, \ldots, in_n] \in ValidOrders(I)$ and σ is a permutation on n elements, then $[in_{\sigma(1)}, \ldots, in_{\sigma(n)}] \in ValidOrders(I)$.*

Proof. Every permutation of a finite set can be expressed as the product of transpositions [92] (p.60). Let m be the number of such transpositions for σ. We prove the result by induction on m.

If $m = 0$, then σ is the identity and we have by hypothesis:

$$[in_{\sigma(1)}, \ldots, in_{\sigma(n)}] \in ValidOrders(I)$$

If $\sigma = \tau_{m+1} \circ \tau_m \circ \cdots \circ \tau_1$, then we denote $\sigma' = \tau_m \circ \cdots \circ \tau_1$. By induction hypothesis, we have:

$$[in_{\sigma'(1)}, \ldots, in_{\sigma'(n)}] \in ValidOrders(I)$$

Then, by applying Lemma 8 to the transposition $(ij) = \tau_{m+1}$, we have:

$$[in_{\tau_{m+1} \circ \sigma'(1)}, \ldots, in_{\tau_{m+1} \circ \sigma'(n)}] \in ValidOrders(I)$$

Finally, because $\sigma = \tau_{m+1} \circ \sigma'$, we conclude the induction step. □

Since inputs in a valid order of removal steps are always distinct (Lemma 5) and their order does not make a difference (Proposition 2), such orders can simply be seen as sets even if, in practice, for IMPRO and MOCCO, orders of removal steps are implemented as lists. More precisely, two valid orders of removal steps are equivalent if they correspond to the same set of inputs. We leverage this property to reduce the number of orders of removal steps to consider, e.g., since $[in_1, in_2]$ is equivalent to $[in_2, in_1]$, we only have to check that $[in_1, in_2]$ is valid to know whether $[in_2, in_1]$ is valid or not. This means that, when inputs are provided in a given order $[in_1, \ldots, in_n]$, these tools need only to check for valid orders of removal steps in increasing order (hence without backtracking on previous inputs), i.e., $[in_{i_1}, \ldots, in_{i_m}]$, where $1 \leq i_1 < \cdots < i_m \leq n$. We call *canonical orders* such orders of removal steps. For instance, $[in_2, in_4, in_7]$ is in canonical order, but $[in_4, in_2, in_7]$ is not. Thus, to save time when checking the validity of orders of removal steps, we consider only canonical orders.

Theorem 1 (Canonical Order)**.** *Let $I = \{in_1, \ldots, in_n\}$ be an input set. There exists $[in_{i_1}, \ldots, in_{i_m}] \in ValidOrders(I)$ such that $1 \leq i_1 < \cdots < i_m \leq n$ and:*

$$\sum_{1 \leq j \leq m} cost(in_{i_j}) = gain(I)$$

Proof. Let $[in_{i'_1}, \ldots, in_{i'_m}] \in ValidOrders(I)$ be a valid order of removal steps with a maximal cumulative cost i.e., according to our definition of the gain Section 3.3.3:

$$\sum_{1 \leq j \leq m} cost(in_{i'_j}) = gain(I)$$

If $m = 0$, then [ ] satisfies the theorem for a gain = 0.

Otherwise, we assume $m > 0$. According to Lemma 5, $[in_{i'_1}, \ldots, in_{i'_m}]$ contains m distinct inputs. Let $\sigma \in S_m$ be the permutation such that:

$$[in_{\sigma(i'_1)}, \ldots, in_{\sigma(i'_m)}] = [in_{i_1}, \ldots, in_{i_m}]$$

with $1 \leq i_1 < \cdots < i_m \leq n$.

According to Proposition 2, we have $[in_{i_1}, \ldots, in_{i_m}] \in \mathit{ValidOrders}(I)$.

Moreover, because a permutation of the elements of a sum does not change the value of the sum, we have:

$$\sum_{1 \leq j \leq m} \mathit{cost}(in_{i_j}) = \sum_{1 \leq j \leq m} \mathit{cost}(in_{i'_j}) = \mathit{gain}(I)$$

Hence the theorem. □

Theorem 1 allows us to focus on inputs in increasing order, instead of exploring all possible input orders. Thus, this optimization saves computation steps and makes computing the gain more tractable. More precisely, since there are $!k$ ways of ordering k selected inputs, there are "only" $\frac{!n}{!k!(n-k)}$ remaining possibilities to investigate, instead of $\frac{!n}{!(n-k)}$.