Metamorphic Testing for Web System Security

Nazanin Bayati Chaleshtari, Fabrizio Pastore, *Member, IEEE,* Arda Goknil, and Lionel C. Briand, *Fellow, IEEE*

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 behavior (e.g., the response to be received after a specific HTTP GET request).

In this paper, we propose Metamorphic Security Testing for Web-interactions (*MST-wi*), a metamorphic testing approach 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 accompanied 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 relations and collected data.

We provide a catalog of 76 system-agnostic MRs to automate security testing in Web systems. It covers 39% of the OWASP security testing activities not automated by state-of-the-art techniques; further, our MRs can automatically discover 102 different types of vulnerabilities, which correspond to 45% of the vulnerabilities due to violations of security design principles according to the MITRE CWE database. We also define guidelines that enable test engineers to improve the testability of the system under test with respect to our approach.

We evaluated *MST-wi* effectiveness and scalability with two well-known Web systems (i.e., Jenkins and Joomla). It automatically detected 85% of their vulnerabilities and showed a high specificity (99.81% of the generated inputs do not lead to a false positive); our findings include a new security vulnerability detected in Jenkins. Finally, our results demonstrate that the approach scale, thus enabling automated security testing overnight.

Index Terms—System Security Testing, Metamorphic Testing, Domain-specific Languages.

1 Introduction

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, in Web systems, discovering security vulnerabilities (i.e., faults preventing the system from fulfilling its security requirements) is essential [1], [2], [3], [4], [5]. It 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* [6], [7], [8]. 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 [9]. 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 [10], [11], [12], [13].

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 [6]. Furthermore, most of them focus on a particular vulnerability type

• *N.B. Chaleshtari is affiliated with the University of Ottawa, Canada. F. Pastore is with the SnT Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg. A. Goknil is affiliated with SINTEF Digital, Norway; the work of A. Goknil has been partially performed while affiliated with the SnT Centre. L.C. Briand is affiliated with both the University of Luxembourg and University of Ottawa.*
E-mails: n.bayati@uottawa.ca fabrizio.pastore@uni.lu arda.goknil@sintef.no lionel.briand@uni.lu lbriand@uottawa.ca

Manuscript received August 2022; revised January 2022.

(e.g., buffer overflows [14], [15]) 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 [16], [17]. *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* [18]. In MT, system properties are captured as MRs that are used to automatically transform an initial set of test inputs into follow-up test inputs. If the system outputs for the initial and follow-up test inputs violate the corresponding MR, it is concluded that the system is faulty.

Considerable research has been devoted to developing MT approaches for application domains such as computer graphics (e.g., [19], [20], [21], [22]), Web services (e.g., [23], [24], [25]), and embedded systems (e.g., [26], [27], [28], [29]). Unfortunately, only a few approaches target security aspects [30]; also, their applicability is limited to the functional testing of security components (e.g., code obfuscators [30]) or the verification of specific security bugs (e.g., heartbleed [31]). They do not support the specification of general security properties by using MRs. Although MT is automatable, few MT approaches provide proper tool support [18].

Our goal in this paper is to use MT to both automatically generate test inputs for security testing and address 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 would 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. 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 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 [32], which facilitates the specification of MRs.
- A catalog of system-agnostic MRs targeting 102 security vulnerabilities of 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 [33] to derive source inputs automatically from the system under test. It provides an MT algorithm integrated into the JUnit framework [34] as part of the testing framework. The algorithm performs MT based on the executable MRs in Java and the 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 addressing specific vulnerability types.
- The applicability of *MST-wi* to discover common and important security vulnerability types described in the Common Weaknesses Enumeration Repository [36]. Our results show that *MST-wi* enables testing for 101 (45%) vulnerability types due to errors in applying security design principles.
- 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 [37], and Joomla, a content management system [38]. *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 paper largely extends our previous conference paper [39] published at the 13th IEEE International Conference on Software Testing, Verification and Validation (ICST'20).

1. OWASP (Open Web Application Security Project) is one of the best known organizations that focus on software security [35].

An earlier version of our tool was demonstrated [40] at the 42nd International Conference on Software Engineering (ICSE'20). This paper brings together, refines, and significantly extends the ideas from the above papers:

- We extend our DSL with additional data representation and utility functions.
- We extend our 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 [41], the ones belonging to the CWE Top-25 most dangerous software errors [42], and the ones in the OWASP Top-10 Web security risks [43].
- 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 (the latter being entirely new). Our results include the discovery of a new security vulnerability in Jenkins that received a CVE identifier [44].

Our toolset [45], [46], our catalog of MRs [47], and empirical data [48] are publicly available.

This paper is structured as follows. Section 2 provides the background information regarding MT. In Section 3, we present an overview of the approach. Sections 4 to 7 describe the core technical solutions. Section 8 describes our catalog of MRs. In Section 9, we investigate the security vulnerability types that *MST-wi* can address and the testability guidelines for *MST-wi* to address these vulnerabilities. Section 10 reports on the results of the empirical validation conducted with two open-source case studies. Section 11 addresses threats to validity. Section 12 discusses the related work. We conclude the paper in Section 13.

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 [49]. 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* [18].

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. 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:

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

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.

3 Overview of the Approach

The process in Fig. 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.

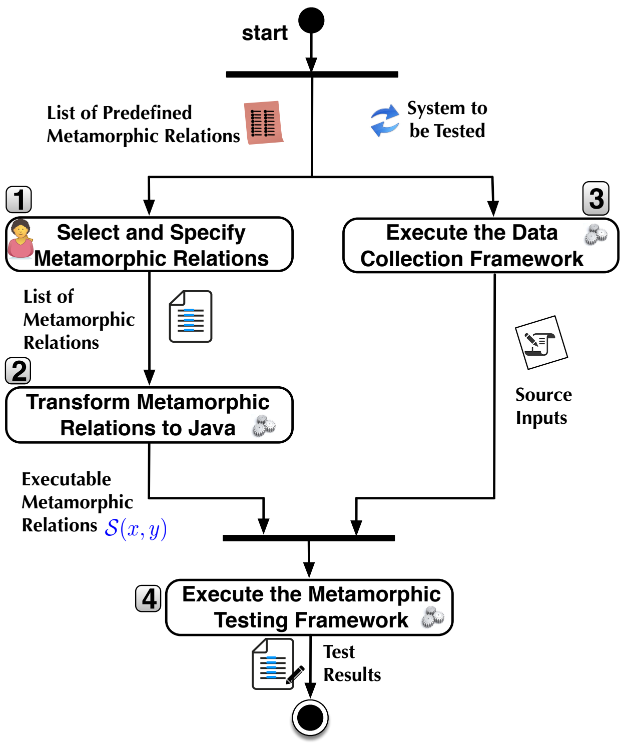

start
List of Predefined Metamorphic Relations
System to be Tested
1
Select and Specify Metamorphic Relations
3
Execute the Data Collection Framework
List of Metamorphic Relations
2
Transform Metamorphic Relations to Java
Source Inputs
Executable Metamorphic Relations $\mathcal{S}(x, y)$
4
Execute the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

Fig. 1. Overview of the approach.

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 with input interfaces not supported yet (e.g., Silverlight plug-ins [50]) and to load data collected by new data collection methods.

Sections 4 to 7 explain the details of each step in Fig. 1, with a focus on how we achieved our automation objectives.

4 Step 1: Select and Specify MRs

Step 1 in Fig. 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 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.

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 7).

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.

4.1 SMRL Grammar

SMRL is an extension of Xbase [51], an expression language provided by Xtext [52]. Xbase specifications can be translated to Java programs and compiled into executable Java bytecode.

We rely on Xbase since DSLs extending Xbase inherit the syntax of a Java-like expression language and language infrastructure components, including a parser, a linker, a compiler, and an interpreter [51]. Further, it provides features common in modern programming languages, including lambda expressions, type inference, and simple operator overloading. These features will facilitate the adoption of SMRL.

SMRL extends Xbase by introducing (i) data representation functions, (ii) boolean operators to specify security properties, and (iii) Web-utility functions to express data properties and transform data. We can extend these functions by defining new Java APIs invoked in MRs.

Fig. 2 presents an MR written in our SMRL editor. The relation checks whether other users can access the URLs dedicated to specific users through a direct request. We use it as a running example in the rest of the paper.

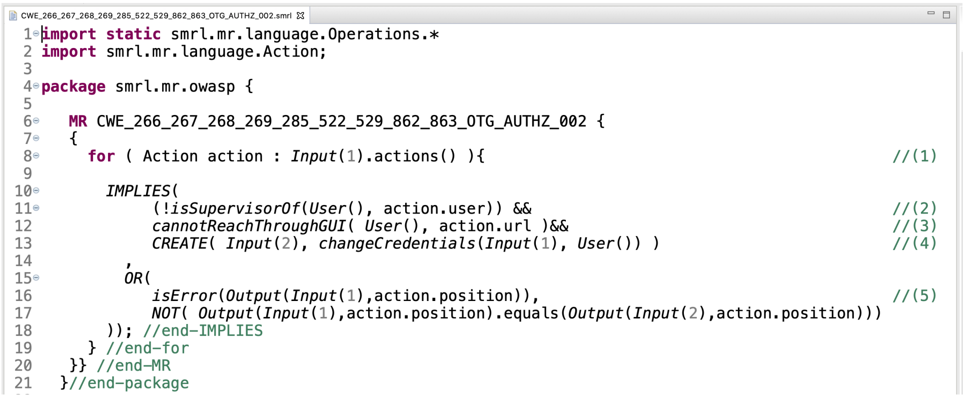
The SMRL grammar extends the Xbase grammar, which extends the Java grammar. Each SMRL specification can rely on external classes and APIs; in practice, it can have have an arbitrary number of import declarations indicating the APIs used in MRs (Line 1 in Fig. 2).

A package declaration resembles the Java package structure and can contain one or more MRs. Line 4 in Fig. 2 declares the package *smrl.mr.owasp*, which is the package for our MRs automating the testing activities in the OWASP testing guidelines [9]. Like in Java, MRs defined in different SMRL specification files can belong to the same package.

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

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:


CWE_266_267_268_269_285_522_529_862_863_OTG_AUTHZ_002.smrl

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

package smrl.mr.owasp {

   MR CWE_266_267_268_269_285_522_529_862_863_OTG_AUTHZ_002 {
   {
     for ( Action action : Input(1).actions() ){                                //(1)

       IMPLIES(
            (!isSupervisorOf(User(), action.user)) &&                            //(2)
            cannotReachThroughGUI( User(), action.url )&&                        //(3)
            CREATE( Input(2), changeCredentials(Input(1), User()) )             //(4)
         ,
         OR(
            isError(Output(Input(1),action.position)),                          //(5)
            NOT( Output(Input(1),action.position).equals(Output(Input(2),action.position)))
       )); //end-IMPLIES
     } //end-for
   }} //end-MR
  }//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.

Fig. 2. An MR for the Bypass Authorization Schema vulnerability.

TABLE 1
Data functions in SMRL.

Category	Data function	Description
Interaction	Input(int i)	Returns the ith input sequence.
	Action(int i)	Returns the ith input action.
	ActionAvailable-WithoutLogin(int i)	Returns the ith input action that can be performed without logging into the system.
	User(int i)	Returns the ith user of the system.
	ParameterValueUsedBy-OtherUsers(Action i, par i)	Returns, for the same action and parameter position, the ith 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 subfolder, ignoring images, and replacing symbolic links (e.g., ‘plugins’ is mapped to ‘plugin’ in Jenkins).
	RandomAdminFilePath(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 nth 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.

- *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 1 presents the data functions in SMRL, grouped by category. Each data function returns a data class instance.

Fig. 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

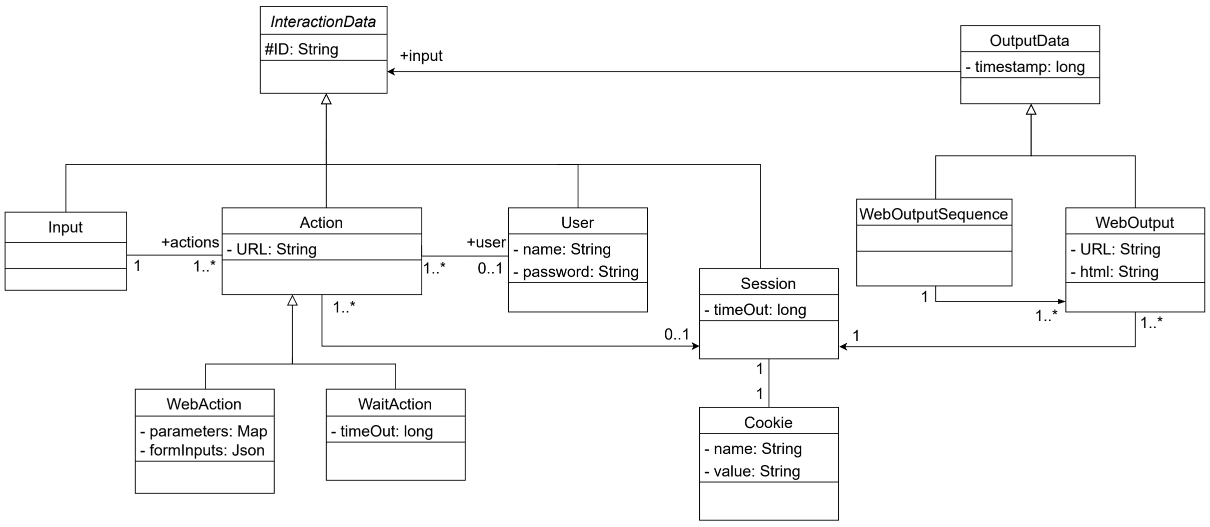


Fig. 3. Metamorphic data classes in SMRL.

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 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 paper, 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.

4.3 MR Operators

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

The MR in Fig. 2 includes a metamorphic expression using the operator `IMPLIES`. Since the expression is within a loop body, the relation holds only if the expression evaluates to true in all the iterations over the input actions.

The semantics of 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, operator `CREATE` defines the follow-up input `Input(2)` as a modified copy of `Input(1)`. 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 can be 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.

4.4 Web-Utility Functions

MRs for security testing often capture complex properties of Web systems that we cannot express with simple boolean

TABLE 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.

or arithmetic operators. Therefore, SMRL provides some functions that capture standard Web system properties and alter Web data. Table 2 describes a portion of the 55 Web-specific functions in SMRL [53]. Each one is provided as a method of the SMRL API. Engineers can specify additional functions as Java methods. The new functions can be used in SMRL thanks to the underlying Xtext framework.

The MR in Fig. 2 uses the Web-specific functions `cannotReachThroughGUI`, `isSupervisorOf`, `isError`, and `changeCredentials`. The relation indicates that the same sequence of actions should provide different outputs when performed by two 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) 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 output that should be observed for `User(2)`, which may be the same error, a different error, or an empty page.

Function `cannotReachThroughGUI` in Fig. 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.

5 Step 2: Transform MRs to Java

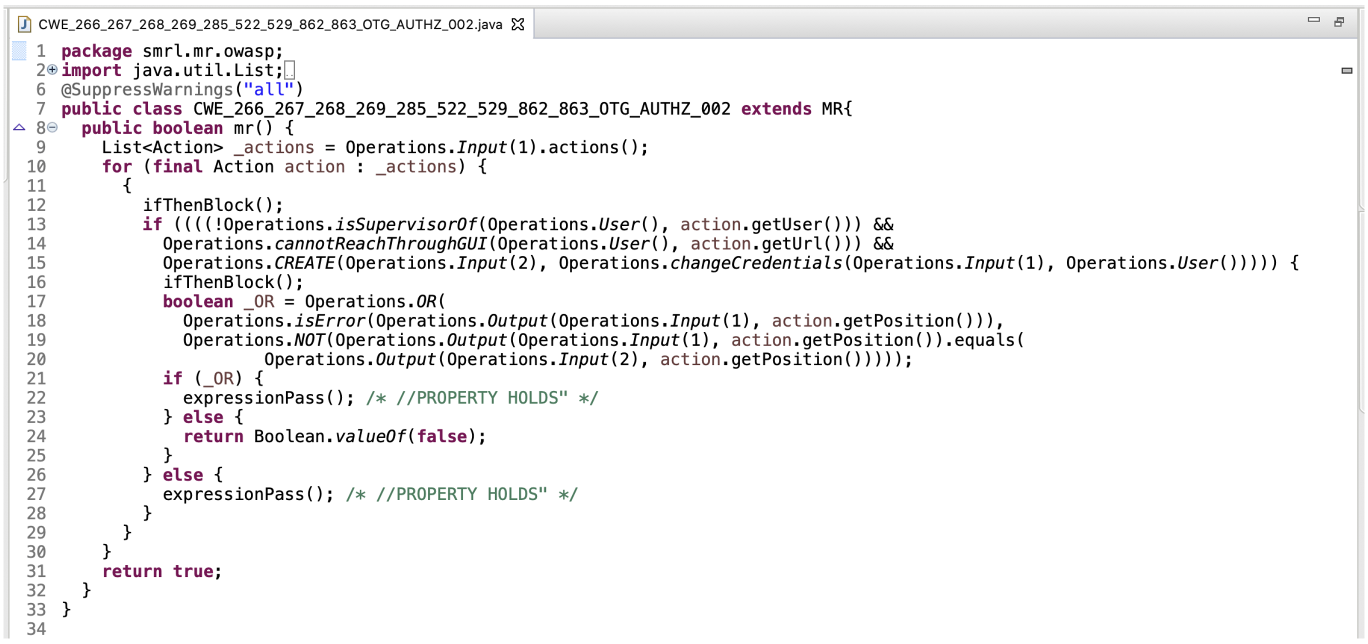
In Step 2, SMRL specifications are automatically transformed into Java code. To this end, we extended the Xbase compiler (hereafter, SMRL compiler). Each MR is transformed into a Java class with the relation name and package. The generated classes extend class `MR` and implement its method `mr`.

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

Fig. 4 shows the Java code generated from the MR in Fig. 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.

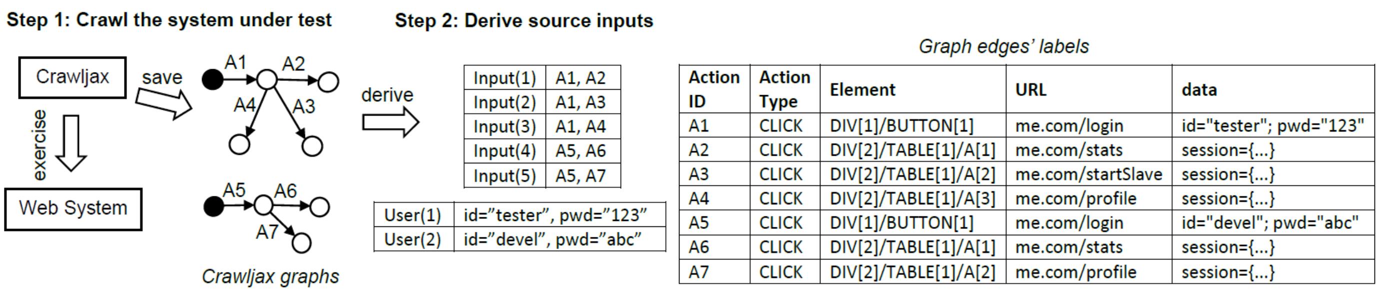
6 Step 3: Data Collection

In Step 3, we rely on an extended version of the Crawljax Web crawler to automatically derive source inputs [54], [55]. 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 [56]. 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



```java
package smrl.mr.owasp;
import java.util.List;
@SuppressWarnings("all")
public class CWE_266_267_268_269_285_522_529_862_863_OTG_AUTHZ_002 extends MR{
  public boolean mr() {
    List<Action> _actions = Operations.Input(1).actions();
    for (final Action action : _actions) {
      {
        ifThenBlock();
        if ((((!Operations.isSupervisorOf(Operations.User(), action.getUser())) &&
          Operations.cannotReachThroughGUI(Operations.User(), action.getUrl())) &&
          Operations.CREATE(Operations.Input(2), Operations.changeCredentials(Operations.Input(1), Operations.User())))) {
          ifThenBlock();
          boolean _OR = Operations.OR(
            Operations.isError(Operations.Output(Operations.Input(1), action.getPosition())),
            Operations.NOT(Operations.Output(Operations.Input(1), action.getPosition()).equals(
                    Operations.Output(Operations.Input(2), action.getPosition()))));
          if (_OR) {
            expressionPass(); /* //PROPERTY HOLDS" */
          } else {
            return Boolean.valueOf(false);
          }
        } else {
          expressionPass(); /* //PROPERTY HOLDS" */
        }
      }
    }
    return true;
  }
}
```

Fig. 4. Java code generated from the MR in Fig. 2.



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

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

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

Fig. 5. Data collection with a simplified example.

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.

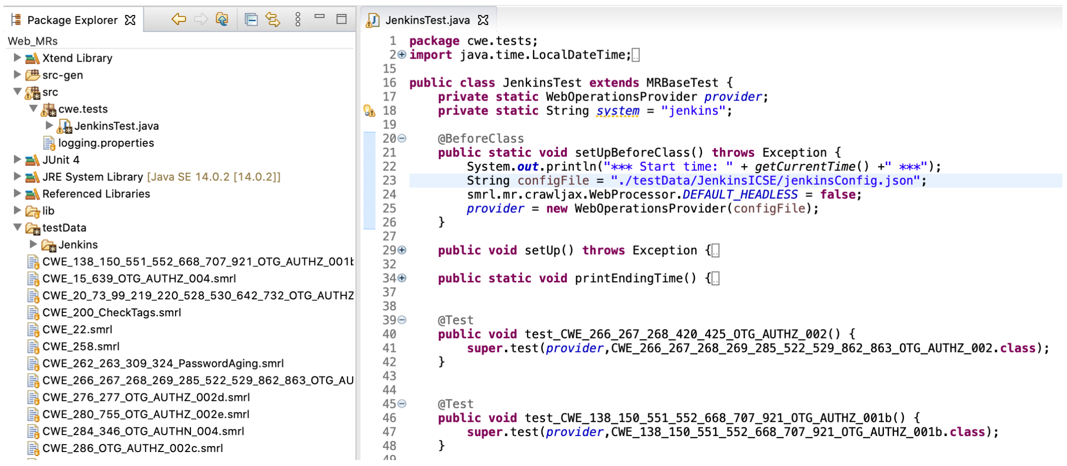
Our Crawljax extensions enable replicating and modifying portions of a crawling session, which is necessary to simulate attacks (e.g., by changing the URL accessed within an input sequence). To this end, in addition to (i) the Crawljax actions and (ii) the XPath of the elements targeted by the actions (e.g., a button clicked on), our extension records (iii) the URLs requested by the actions, (iv) the data in the HTML forms, and (v) the background URL requests. This additional data enables, for example, replicating modified portions of crawling sessions that request URLs not appearing on the last Web page returned by the system. To crawl the SUT, we require only its URL and a list of credentials.

Fig. 5 exemplifies the data collection steps. First, Crawljax generates the graphs of the system under test. Second, our toolset automatically derives source inputs from the graphs. A source input is a path from the root to a leaf of a Crawljax graph in a depth-first traversal. Third, the SMRL functions query the source inputs (see Section 7). For example, $\texttt{Input}(\texttt{i})$ returns the i^{th} input sequence; $\texttt{User}(\texttt{i})$ returns the i^{th} unique login credentials in the input sequences.

In addition to Crawljax, *MST-wi* processes manually implemented test scripts to generate additional source inputs. It processes scripts based on the Selenium framework [57] and derives a source input from each script. We rely on test scripts to exercise complex interaction sequences not triggered by Crawljax (see Section 10). Crawljax performs an almost exhaustive exploration of the Web interface, which is typically not achieved by test scripts. Engineers can reuse scripts developed for functional testing or define new ones.

7 Step 4: Execute the MT Framework

We automatically perform testing based on the executable MRs in Java and the data collected by the data collection framework (Step 4 in Fig. 1). Our testing framework relies on the JUnit framework [34] to



```
Web_MRs
▶ Xtend Library
▶ src-gen
▼ src
  ▼ cwe.tests
    ▶ JenkinsTest.java
    logging.properties
▶ JUnit 4
▶ JRE System Library [Java SE 14.0.2 [14.0.2]]
▶ Referenced Libraries
▶ lib
▼ testData
  ▶ Jenkins
  CWE_138_150_551_552_668_707_921_OTG_AUTHZ_001b
  CWE_15_639_OTG_AUTHZ_004.smrl
  CWE_20_73_99_219_220_528_530_642_732_OTG_AUTHZ
  CWE_200_CheckTags.smrl
  CWE_22.smrl
  CWE_258.smrl
  CWE_262_263_309_324_PasswordAging.smrl
  CWE_266_267_268_269_285_522_529_862_863_OTG_AU
  CWE_276_277_OTG_AUTHZ_002d.smrl
  CWE_280_755_OTG_AUTHZ_002e.smrl
  CWE_284_346_OTG_AUTHN_004.smrl
  CWE_286_OTG_AUTHZ_002c.smrl
```

```java
 1 package cwe.tests;
 2⊕import java.time.LocalDateTime;
15
16 public class JenkinsTest extends MRBaseTest {
17     private static WebOperationsProvider provider;
18     private static String system = "jenkins";
19
20⊖    @BeforeClass
21     public static void setUpBeforeClass() throws Exception {
22         System.out.println("*** Start time: " + getCurrentTime() +" ***");
23         String configFile = "./testData/JenkinsICSE/jenkinsConfig.json";
24         smrl.mr.crawljax.WebProcessor.DEFAULT_HEADLESS = false;
25         provider = new WebOperationsProvider(configFile);
26     }
27
29⊕    public void setUp() throws Exception {
32
34⊕    public static void printEndingTime() {
37
38
39⊖    @Test
40     public void test_CWE_266_267_268_420_425_OTG_AUTHZ_002() {
41         super.test(provider,CWE_266_267_268_269_285_522_529_862_863_OTG_AUTHZ_002.class);
42     }
43
44
45⊖    @Test
46     public void test_CWE_138_150_551_552_668_707_921_OTG_AUTHZ_001b() {
47         super.test(provider,CWE_138_150_551_552_668_707_921_OTG_AUTHZ_001b.class);
48     }
```

Fig. 6. An example JUnit test case to select and execute MRs.

integrate MT into traditional testing environments. To choose MRs to be executed, the engineer writes a JUnit test case. Fig. 6 presents a JUnit test case selecting multiple MRs (i.e., CWE_266.._OTG_AUTHZ_002, and CWE_138.._OTG_AUTHZ_001b) through function `test`. MRs need to be copied into the workspace (see the files with extension `.smrl` in the project explorer window in Fig. 6) and referred to in the JUnit test case (see JenkinsTest.java in Fig. 6). *MST-wi* provides a Java class, `MRBaseTest` (Line 16 in Fig. 6), which extends the JUnit framework with utility functions to facilitate the selection of MRs. Method `setUpBeforeClass` is used to specify the configuration for `WebOperationsProvider` (Lines 21-26), i.e., the component in charge of loading source inputs.

MRs are executed as standard JUnit test classes through the Eclipse user interface or the console wrapper. Each MR is treated as a distinct JUnit test case (Lines 40-48 in Fig. 6). For each MR, class `MRBaseTest` automatically invokes our MT algorithm that executes the MR.

Fig. 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 [58]. 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.

Function `iterateOverInputTypes` is driven by the methods exposed by the data provider (Lines 5 and 6). The data provider works as a circular array that provides, in each iteration of `iterateOverInputTypes`, a different view of the collected data. 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

```
Require: MR, the bytecode of the metamorphic relation to be executed
Require: dataProvider, an object that exposes the data collected by the crawlers
Ensure: Failures, a list of failing executions with contextual information
 1: 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.lenght) 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
13:         end if
14:     end while
15: end function
```

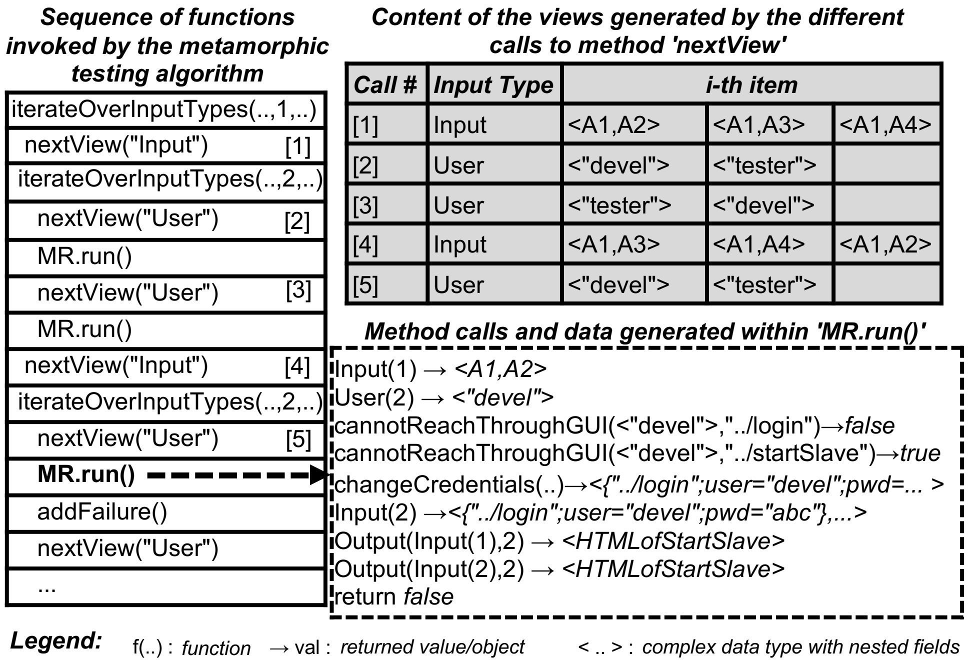
Fig. 7. Metamorphic testing algorithm.

inputs at each invocation of operator `EQUAL`. For example, operator `EQUAL` in Fig. 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 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 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. 8 illustrates the execution of the relation in Fig. 2. The table on the left represents the sequence of functions invoked by our algorithm. In this example, two views for `User` are inspected for each view of `Input`. The first two invocations of `MR.run` return true (not shown in Fig. 8) because the *login* and *stats* pages have been accessed by both users *devel* and *tester*, and, thus, the implication holds. The


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

Content of the views generated by the different calls to method 'nextView'

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

Method calls and data generated within 'MR.run()'

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

Fig. 8. Example data processing for the relation in Fig. 2.

third invocation of `MR.run` returns false because the output page for the *startSlave* URL is the same for the two input sequences and, thus, the relation does not hold. We rely on edit distance to determine if Web pages are equal.

8 Catalog of Metamorphic Relations

We derived a catalog of MRs from the activities described in the OWASP book on security testing [9] and from a set of security vulnerability types related to violations of security design principles and reported in the Common Weakness Enumeration (CWE) database [59].

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. Using this suggestion, we defined the MR in Fig. 2.

The CWE vulnerability types considered for our catalog concern violations of security design principles. As further discussed in Section 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 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 [60]. 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 [6] (e.g., crashes).

We present some of the MRs in our catalog in Figs. 10 to 20. The entire catalog is available for download [53]. All the MRs in our catalog follow the template in Fig. 9 in Section 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 8.2. The identification of patterns facilitates the definition of solutions to a problem [61]; 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

```
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:   )
```

Fig. 9. Template common to all the MRs in our catalog.

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).

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. 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. 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. 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. 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. 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 3 provides an overview of possible template element instances.

TABLE 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

8.1.1 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 not to test the same URL twice (regardless of the user performing the action) in MRs that do not address 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. 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.

8.1.2 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.

```
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.

Fig. 10. CWE_287a_425_OTG_AUTHN_001: Testing for credentials transported over an encrypted channel

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). 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. 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. 13). 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 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. 14). 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. 15); 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. 16), 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. 10. In 57 MRs

```
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.

Fig. 11. CWE_302_471_472_784_807: Testing for assumed-immutable elements

of our catalog, the follow-up inputs perform the same sequence of actions as the source input but with at least one parameter modification.

8.1.3 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* than those of the source inputs. For instance, in MR `CWE_266_..._OTG_AUTHZ_002` (Fig. 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. 17) replaces URL parameters (e.g., an ID) with paths of files on the SUT. Similarly, `CWE_15_639_OTG_AUTHZ_004` (Fig. 18) 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.

```
 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).

Fig. 12. CWE_79_a_XSSreflected: Testing for reflected cross-site scripting

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 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.

Two patterns occur with the same frequency (i.e., *P1* and *P2* in Table 4); we present them in the order they appear in Table 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. 19), 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 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.

Fig. 13. CWE_286_OTG_AUTHZ_002c: Testing for incorrect user management

```
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.

Fig. 14. CWE_841: Testing for improper enforcement of behavioral workflow

MR of this pattern is MR CWE_792_793_794_795_796_797_A (Fig. 20).

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. 15, 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 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.

Fig. 15. OTG_SESS_003: Testing for session fixation

```
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.

Fig. 16. CWE_262_263_309_324: Testing for password aging

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, require-

```
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.

Fig. 17. CWE_20_73_99_219_220_528_530_642_732_OTG_AUTHZ_001a: Testing for directory traversal

TABLE 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

ments, or design document) supports its testing [62]. 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*.

9.1 Targeted Vulnerabilities and Testing Activities

To address our first research question, we study the security testing activities recommended by OWASP [35]. These activities have been described as part of the OWASP testing guidelines Version 4.0 [9] 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

```
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.

Fig. 18. CWE_15_639_OTG_AUTHZ_004: Testing for externally controlled elements

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 [63]. 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 [59]. We provide the following definitions of *vulnerability* and *weakness* since their definitions in the CWE framework [64] 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* [41], *software development concepts* [65], *research concepts* [66], *software fault patterns* [67], *most dangerous errors* [42], and *hardware design* [68]. Other views map the weaknesses to some security-related catalogs (e.g., OWASP Top 10 [43] and SERT CEI C Coding standards [69]).

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 [70], [71]. 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 [41]. 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 [72] 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 [41], the weaknesses in the view for the CWE Top 25 most dangerous software errors (CWE Top 25) [42], and the weaknesses in the view for the OWASP Top 10 Web security risks (OWASP Top 10) [73]. 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 [64]'.

```
 MR CWE_94_96_B {
{
  keepDialogsOpen = true;                                                    //(1)
  for (Action action : Input(1).actions()){                                  //(2)
     var pos = action.getPosition();                                         //(3)
     for (var x=0; action.containFormInput() && x < action.formInputs.size; x++){  //(4)
        IMPLIES(
            action.isClickOnButton &&                                        //(5)
            notTried(x+action.url, Input(1).actions().get(pos).getElementURL()) && //(6)
            userCanRetrieveContent(action.getUser(), Output(Input(1),pos)) &&      //(7)
            CREATE(Input(2), Input(1) ) &&                                   //(8)
            Input(2).actions.get(pos).setFormInput(x, StaticInjectionString())     //(9)
            ,
        OR(                                                                  //(10)
            isError(Output(Input(2), pos)),
            userCanRetrieveContent(action.getUser(), Output(Input(2),pos)))
     );//end-IMPLIES
    }//end-for parameter
   }//end-for action
  }
 }//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 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.

Fig. 19. CWE_94_96_B: Testing for static code injection

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* [69] since they mainly focus on software implementation. We ignore the CWE view for *hardware design* since *MST-wi* does not address hardware vulnerabilities.

In our analysis, we include the CWE Top 25 and OWASP Top 10 views to assess to what extent *MST-wi* can address the most widespread and critical security vulnerabilities. The CWE Top 25 view lists twenty-five most widespread weaknesses which are often easy to find and exploit. These weaknesses are considered dangerous because they typically *allow attackers to completely take over the control of software, steal data, or prevent software from working [42].* The OWASP Top 10 [43] is the list of the ten most common web application security risks edited by the Open Web Application Security Project [74], 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 [73].

9.2 RQ1: Oracle Problem

9.2.1 Analysis Procedure

To respond to RQ1, we study the security testing activities recommended by OWASP [9]. 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 [75]. 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. 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*.

9.2.2 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 5, for each OWASP testing category, some of its security testing activities along with oracle automation strategies.

```
 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 dialogues 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.

Fig. 20. CWE_792_793_794_795_796_797_A: Testing for incomplete filtering of one or more instances of special elements

TABLE 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

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* [76], 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 [76].

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 [77]. 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 [6]. For instance, testing activity *Testing for Buffer Overflow* in Table 5 is automated by looking for system crashes in response to lengthy inputs [78]. One of the testing methods in the activity is testing for the format string [79]. 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 [80]. 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 [81] 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* [82]. Though it enables engineers to compare the content of site maps [83] 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 [84]. With *MST-wi*, engineers, instead, can focus on the specifications of system-level properties without performing such manual testing activities. 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 [85]. 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. 17).

Table 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 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 (8% and 2%, respectively). The limited applicability of implicit oracles shows that most of the solutions relying on them (e.g., fuzz testing tools [86]), 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 paper. 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*

TABLE 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%	8%	21%	28%	24%	18%

*Details are available online [53]. 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 [9].

Direct Object References in Table 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. 15. A session fixation vulnerability occurs when the system under test does not renew its session cookie(s) after successful user authentication [87]. `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*).

9.3 RQ2: Vulnerability Types

9.3.1 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 9.1 (i.e., *Common security 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) [88] associated with the weakness. Based on the information collected from our inspection, we implemented, using SMRL, a new MR that address 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 [48].

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 [89]), 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 addressed 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 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 7, a subset of the weaknesses in the CWE view for common security architectural tactics. We refer to Table 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 addressed 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. [90] 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 [91] and Sonarqube [92] for DAST and SAST, respectively) and

TABLE 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

TABLE 8
Summary of the CWE architectural security design principles and weaknesses addressed by *MST-wi*.

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

TABLE 9
Summary of the CWE Top 25 weaknesses addressed by *MST-wi*.

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

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 addressed 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., address weaknesses not discovered by these tools).

9.3.2 Results

Table 8 presents a summary of the CWE security design principles and related security weaknesses addressed by *MST-wi*. The first column in Table 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 addressed 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 [48]. If we sort security design principles based on the percentage of weaknesses addressed 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

TABLE 10
Summary of the security weaknesses for OWASP Top 10 security risks addressed by *MST-wi*.

OWASP Security Risk	**Weaknesses**		**Addressed 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%)

TABLE 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.

principles are *Manage User Sessions* (1st for generic weaknesses, 2nd considering all the weaknesses) and *Validate Inputs* (2nd for generic weaknesses, 1st 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 (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 ranking 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 addresses 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.

TABLE 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

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

Table 9 gives a summary of the CWE Top 25 weaknesses addressed 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 address 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 10 presents a summary of the security weaknesses related to the OWASP Top 10 security risks addressed by *MST-wi*. It addresses 64 out of the 126 weaknesses (51%) and 61 out of the 117 generic weaknesses (52%) in this view. It addresses 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 authenti-

TABLE 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

cation. All involve malicious user-system interactions.

MST-wi can address 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 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 11). Also, some weaknesses can be discovered only through program analysis, which *MST-wi* does not support (see R2 in Table 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 [93]. Example cases are reported in Table 7 and concern log files omitting or 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 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 [94]).

TABLE 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

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 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 11). This is the case for *Improper Validation of Certificate with Host Mismatch* (see Table 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 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 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 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

TABLE 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

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 13 and 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.

Tables 16 to 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 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 addressed by any other* in Table 16), while *MST-wi* addresses 101 weaknesses (see Column *Weaknesses addressed by* MST-wi). Concerning complementarity, we observe that *MST-wi* can detect 56 weaknesses that any other approach cannot address (see column *Weaknesses addressed by MST but not by Any other*). Most of the weaknesses addressed 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 addressed by *MST-wi* (see Column *Weaknesses addressed by* MST-wi). Together, all SOTA approaches can address only 39 weaknesses that *MST-wi* does not address (see column *Weaknesses not addressed by MST but addressed by Any other*). The tool addressing most of the weaknesses not addressed by *MST-wi* is SA2, which fares the best in the study of Elder et al. [90]. The design principles in which SOTA tools provide greater benefits (i.e., the number of weaknesses addressed only by *MST-wi* is lower than the number of weaknesses addressed 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 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 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 addressed 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 address 20 weaknesses not discovered by *MST-wi*, while *MST-wi* targets 30 weaknesses not addressed by SA2. However, all the SOTA approaches detect together 29 weaknesses not addressed 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 18), the number of weaknesses addressed by *MST-wi* and the best SOTA approach is similar (15 versus 14); however, they complement each other. *MST-wi* addresses five weaknesses not addressed by other approaches: *CWE-306 (Missing Authentication for Critical Function)*, *CWE-276 (Incorrect Default Permissions)*,

TABLE 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 addressed 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 addressed exclusively by *MST-wi* is higher than the number of weaknesses addressed exclusively by the combined use of other approaches.

Security Design Principle	**Weaknesses addressed by**						**Weaknesses addressed by MST but not by**					**Weaknesses not addressed by MST but addressed 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 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 16.

OWASP Security Risk	**Weaknesses addressed by**						**Weaknesses addressed by MST but not by**					**Weaknesses not addressed by MST but addressed 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 18
Comparison of *MST-wi* with SOTA approaches for the verification of weaknesses in the CWE Top 25 list. Highlights follow the same convention ofTable 16.

Weaknesses addressed by						**Weaknesses addressed by MST but not by**					**Weaknesses not addressed by MST but addressed 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

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 addressed 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 addressed 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 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 address 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 address 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 address 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 addressed. 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.

9.4 RQ3: testability guidelines

9.4.1 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 [95] lists 21 testability factors: *observability*, *controllability*, *complexity*, *cohesion*, *understandability*, *inheritance*, *reliability*, *availability*, *flexibility*, *test suite reusability*, *maintainability*, *unit size*, *statefulness*, *isolateability*, *software process capability*, *modularity*, *test support environment*, *fault-proneness*, *manageability*, *quality of the test suite*, and *self-documentation*. To guide engineers towards the inspection of the proposed testability features for *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.

9.4.2 Results

Table 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 [95]), (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 [95]), 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 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 [96]. *MST-wi* can be applied to identify this weakness when the feature under test is accessible through a URL/path (see TF1 in Table 15). We match this testability feature to testability factor *Controllability*. In weakness *Insufficient Session Expiration* in Table 7, a Web system permits malicious users to reuse old session credentials or session IDs for authorization [97]. *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 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 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.

Table 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 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 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 8). In Table 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 address 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) [95]. 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 20 and 21 present the testability features that enable testing for the weaknesses in the OWASP Top 10 and CWE Top 25 views, respectively. In Table 20, numbers are in line with the ones in Table 19. Indeed, the testability features that are required for testing a higher subset of weaknesses are also TF1 and TF2 in Tables 20. In Table 21, TF2 and TF3 have a significant role in testing the features.

Tables 19, 20, and 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

TABLE 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.

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 addressing more than 10% of the vulnerabilities in Tables 19, 20, and 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.

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 [32]. 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 [48].

10.1 Subjects of the Evaluation

We applied *MST-wi* to Jenkins [37], a leading open source automation server, and Joomla [38], 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 [98]; Joomla is a PHP application that relies on the MySQL RDBMS [99] and the Apache HTTP server [100]. 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 [101] 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 addresses 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. (one is a software fault that received a CVE identifier [44], the other two are related to Jenkins' default configuration). In addition, we considered, for Jenkins, three new vulnerabilities we discovered with *MST-wi*. 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 22). 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 [102]), 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 because it visited all states. 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 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).

TABLE 22
Vulnerabilities considered in our empirical evaluation.

Subject	Ref.	Vuln. Type	Description
Jenkins	[102]	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.
	[103]	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.
	[104]	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.
	[105]	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.
	[106]	CWE_200	Users with Overall/Read permission ar able to access the URL serving agent logs on the UI due to a lack of permission checks.
	[107]	CWE_384	Jenkins does not invalidate the existing session when a user signs up for a new user account. This allows session fixation.
	[108]	CWE_521	Jenkins does not require users to have strong passwords, which makes it easier for attackers to compromise user accounts.
	[109]	CWE_262	Jenkins does not integrate any mechanism for managing password aging; consequently, users aren't incentivized to update passwords periodically.
	[110]	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.
	[111]	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.
	[112]	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	[113]	CWE_863	Inadequate checks on the tags search fields can lead to an access level violation.
	[114]	CWE_200	Inadequate checks allow users to see the names of tags that were either unpublished or published with restricted view permission .
	[109]	CWE_262	Joomla does not integrate any mechanism for managing password aging; consequently, users aren't incentivized to update passwords periodically.

10.2 RQ4: effectiveness

10.2.1 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).

We measured specificity and sensitivity [115]. 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

TABLE 23
Summary of RQ4 results grouped by data collection method.

Case study	Discovered Vulnerabilities	Crawljax		Crawljax & Manual	
		Specificity	Sensitivity	Specificity	Sensitivity
Jenkins	[102], [103], [104], [105], [106], [107], [108], [109], [110]	99.94%	63.64%	99.90%	81.81%
Joomla	[109], [113], [114]	99.79%	66.67%	99.71%	100.00%
Overall	12	99.87%	64.29%	99.81%	85.71%

the target of HTTP requests with parameters that exercise their vulnerability. Further, we assume that our case study subjects are unlikely to be affected by unknown vulnerabilities (i.e., false negatives) because the selected software versions have been used for years. 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:

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

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:

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

$(1 - Specificity)$ 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.

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 time *MST-wi* reports it.

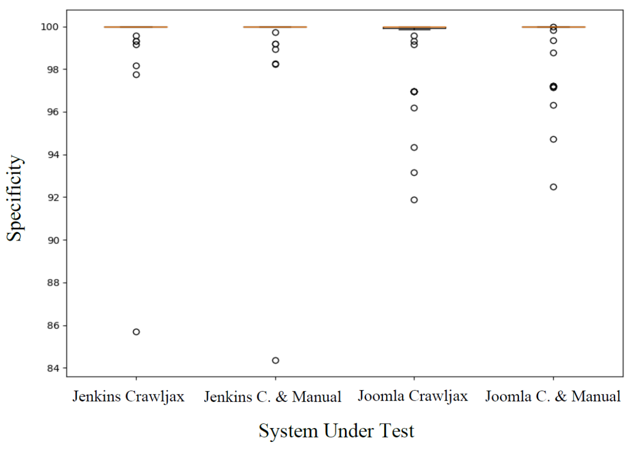


Fig. 21. RQ4: Specificity distribution (values in Table 24)

TABLE 24
RQ4: Specificity distribution

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

10.2.2 Results

Table 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. 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

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

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 [112]. 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 [111], 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 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.

10.3 RQ5: Efficiency

10.3.1 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 paper, we did not perform an empirical evaluation of the efficiency of our

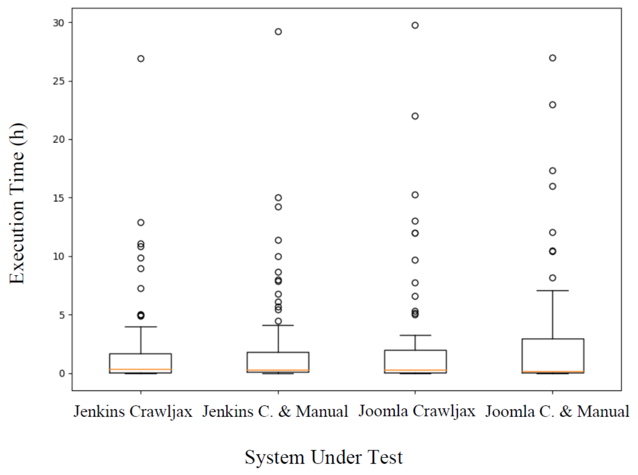


Fig. 22. RQ5. Box-plot reporting execution time in hours (values in Table 25). Some outliers have been removed to ease visualization.

TABLE 25
RQ5: Distribution of execution time (hours)

	Jenkins		Joomla	
	Crawljax	C. & Manual	Crawljax	C. & Manual
First Quarltile	0.06	0.11	0.05	0.084
Second Quarltile	0.38	0.34	0.28	0.22
Third Quarltile	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

crawler (i.e., the extended Crawljax). The interested reader is referred to the original Crawljax publication for further discussions [33]. 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.

10.3.2 Results

Fig. 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 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 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 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 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.

11 Threats to Validity

11.0.1 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 address 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 [116], a security benchmark, thus ensuring that our MRs can discover some of the targeted vulnerabilities (injection vulnerabilities of different kinds, in this case).

11.0.2 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.

TABLE 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

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.

11.0.3 Construct validity

We discuss *construct validity* in terms of face, content, convergent, and predictive validity [117].

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 paper. 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.

About *convergence*, the number of source inputs is weakly correlated to the number of follow-up inputs (i.e., $0.14 \leq$ Spearman's$\rho \geq 0.23$ for all the subjects). The source inputs filtered by MRs' preconditions may explain this weak correlation. The number of follow-up inputs is strongly correlated with the number of actions in follow-up inputs (i.e., Spearman's $\rho \geq 0.85$ for all the subjects), which is expected since we derive follow-up inputs by traversing the crawler graph with DFS.

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.

11.0.4 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 [90]. 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 [39], we did not analyze the Web system developed in the context of the EDLAH2 [118] 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).

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 [119], [120]. 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.

12.1 Software Security Testing

Security testing approaches can be categorized [2] 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 [1]. It is the case for approaches targeting buffer overflows, memory leaks, unhandled exceptions, and denial of service [15], [121], [122], which mostly rely on mutational fuzzing [60], i.e., the generation of new inputs through the random modification of existing ones. For instance, Bekrar et al. [121] 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 [123], [124], [125], [126], [127], [128], [129], [175]. 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 [175]. 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. [130] 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 address 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 9.

12.2 Model-based Security Testing

Model-based approaches [119], [176] mostly target security vulnerability testing (e.g., References [136], [138], [139], [140], [141], [142], [143], [145], [146], [147], [177], [178], [179], [180]), whereas some solutions address security functional testing (e.g., References [131], [132], [133], [134], [135], [137], [148]). Most of them only generate test sequences from security models and do not address the oracle problem. For instance, Marback et al. [141] 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 [146], [147], [148] 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. [146], [147] 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 [148]. Overall, these approaches do not free engineers from significant implementation effort since they require the manual implementation of the executable oracle code.

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., [19], [20], [21], [22]), simulation (e.g., [152], [153], [154]), Web services (e.g., [23], [24], [25]), embedded systems (e.g., [26], [27], [28], [29]), compilers (e.g., [156], [181]), variability and decision support (e.g., [158], [159], [160], [182]), bioinformatics (e.g., [161], [162], [183]), numerical programs (e.g., [163], [164]), and machine learning (e.g., [165], [166]). However, very little attention has been paid to its application in security testing [18].

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 [170], verifying the output of code obfuscators and the rendering of login interfaces [30]), and the verification of low-level properties broken by specific security bugs (e.g., the heartbleed bug [31] 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

TABLE 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. [15]	$-$	$-$	NA	NA	NA	NA	NA
	Bekrar et al [121]	$+$	$-$	NA	NA	NA	NA	NA
	Takanen et al. [122]	$+$	$-$	NA	NA	NA	NA	NA
	Kals et al. [123]			NA	NA	NA	NA	NA
	Martin et al. [124]	$-$	$+$	NA	NA	NA	NA	NA
	Bau et al. [125]	$-$	$+$	NA	NA	NA	NA	NA
	Appelt et al. [126]	$-$	$+$	NA	NA	NA	NA	NA
	Salas et al. [127]	$-$	$+$	NA	NA	NA	NA	NA
	Tripp et al. [128]	$-$	$+$	NA	NA	NA	NA	NA
	Appelt et al. [129]	$-$	$+$	NA	NA	NA	NA	NA
	Huang et al. [130]	$-$	$+$	NA	NA	NA	NA	NA
Model-based Security Testing	Le Traon et al. [131]	NA	$+$	$-$	$-$	NA	NA	NA
	Mouelhi et al. [132], [133]	NA	$+$	$-$	$-$	NA	NA	NA
	Martin et al. [134]	NA	$+$	$-$	$-$	NA	NA	NA
	Martin et al. [135]	NA	$+$	$-$	$-$	NA	NA	NA
	Wimmel and Jürjens [136]	NA	$+$	$-$	$-$	NA	NA	NA
	Masood et al. [137]	NA	$+$	$-$	$-$	NA	NA	NA
	Bertolino et al. [138]	$+$	$+$	$-$	$-$	NA	NA	NA
	Blome et al. [139]	$+$	$-$	$-$	$+$	NA	NA	NA
	He et al. [140]	$+$	$+$	$-$	$-$	NA	NA	NA
	Marback et al. [141]	$+$	$+$	$-$	$-$	NA	NA	NA
	Jürjens et al. [142]	$+$	$+$	$-$	$-$	NA	NA	NA
	Xu et al. [143]	$+$	$+$	$-$	$-$	NA	NA	NA
	Lebeau et al. [144]	$+$	$+$	$-$	$-$	NA	NA	NA
	Whittle et al. [145]	$+$	$+$	$-$	$-$	NA	NA	NA
	Xu et al. [146], [147]	$+$	$+$	$+$	$-$	NA	NA	NA
	Xu et al. [148]	NA	$+$	$+$	$-$	NA	NA	NA
Metamorphic Testing	Mayer and Guderlei [19]	NA	$+$	NA	$+$	$-$	$-$	$-$
	Just and Schweiggert [21]	NA	$+$	NA	$+$	$-$	$-$	$-$
	Kuo et al. [22]	NA	$+$	NA	$+$	$-$	$-$	$-$
	Jameel et al. [149]	NA	$+$	NA	$+$	$-$	$-$	$-$
	Chan et al. [150], [151]	NA	$+$	NA	$+$	$-$	$-$	$-$
	Chen et al. [152]	NA	$+$	NA	$+$	$-$	$-$	$-$
	Ding et al. [153]	NA	$+$	NA	$+$	$-$	$-$	$-$
	Murphy et al. [154]	NA	$+$	NA	$+$	$-$	$-$	$-$
	Sim et al. [155]	NA	$+$	NA	$+$	$-$	$-$	$-$
	Chan et al. [23]	NA	$+$	NA	$+$	$-$	$-$	$-$
	Sun et al. [24]	NA	$+$	NA	$+$	$-$	$-$	$+$
	Zhou et al. [25]	NA	$+$	NA	$+$	$-$	$-$	$+$
	Tse et al. [26]	NA	$+$	NA	$+$	$-$	$-$	$-$
	Chan et al. [27]	NA	$+$	NA	$+$	$-$	$-$	$-$
	Kuo et al. [28]	NA	$+$	NA	$+$	$-$	$-$	$-$
	Jiang et al. [29]	NA	$+$	NA	$+$	$-$	$-$	$-$
	Tao et al. [156]	NA	$+$	NA	$+$	$-$	$-$	$+$
	Yao et al. [157]	NA	$+$	NA	$+$	$-$	$-$	$-$
	Segura et al. [158], [159]	NA	$+$	NA	$+$	$-$	$-$	$+$
	Kuo et al. [160]	NA	$+$	NA	$+$	$-$	$-$	$-$
	Chen et al. [161]	NA	$+$	NA	$+$	$-$	$-$	$-$
	Pullum et al. [162]	NA	$+$	NA	$+$	$-$	$-$	$-$
	Chen et al. [163]	NA	$+$	NA	$+$	$-$	$-$	$-$
	Aruna and Prasad [164]	NA	$+$	NA	$+$	$-$	$-$	$-$
	Xie et al. [165]	NA	$+$	NA	$+$	$-$	$-$	$-$
	Murphy et al. [166]	NA	$+$	NA	$+$	$-$	$-$	$-$
	Segura et al. [167]	NA	$+$	NA	$+$	$-$	$-$	$+$
	Segura et al. [168], [169]	NA	$+$	NA	$+$	$-$	$+$	$-$
	Chen et al. [30]	$-$	$+$	NA	$+$	$+$	$-$	$-$
	Sun et al. [170]	$-$	$+$	NA	$+$	$+$	$-$	$-$
	Luu et al. [171]	NA	$+$	NA	$+$	$-$	$-$	$-$
	Lascu et al. [172]	NA	$+$	NA	$+$	$-$	$-$	$+$
	Ayerdi et al. [173]	NA	$+$	NA	$+$	$-$	$-$	$+$
	Xu et al. [174]	NA	$+$	NA	$+$	$-$	$-$	$-$

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 [18]. For instance, Kuo et al. [28] report on a case study using metamorphic testing to detect faults in a wireless metering system. Ding et al. [153] present a case study for fault detection in a Monte Carlo modeling program simulating photon propagation. Chen et al. [163] 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 [18]. Those which require that MRs be defined either as Java methods [184] or method pre-/post-conditions [185] 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 4). Segura et al. [168], [169] 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. [186] forms the basis for *general metamorphic relations*, which are analogous to metamorphic relation patterns. Zhou et al. [187] 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)* [187] and *metamorphic relation output pattern (MROP)* [167]. 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. [167] 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. [188] 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. [187] 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.

12.4 Summary

In Table 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. [15] 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. [15], as depicted in Table 27. Most automated security testing approaches do not address the oracle problem [123], [124], [125], [126], [127], [128], [129], [175] or rely on an implicit test oracle [15], [121], [122]. The few approaches that do address the oracle problem focus on a limited number of security vulnerabilities [130]. Also, most model-based security testing approaches do not address the oracle problem since they only generate test sequences from security models [136], [141], [144], [145]. Some model-based approaches derive test cases with oracles [146], [147], [148] 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.

13 Conclusion

In this paper, we presented an approach, *MST-wi*, that enables engineers to specify metamorphic relations (MRs) 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 [36], (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 [53].

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.

ACKNOWLEDGMENTS

This work has been carried out as part of the COSMOS Project, which has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 957254. This work was also supported by NSERC of Canada under the Discovery and CRC programs. The authors would like to thank Xuan Phu Mai for his preliminary investigation of CWE vulnerabilities and for setting up the Joomla case study subject.

REFERENCES

[1] C. Haley, R. Laney, J. Moffett, and B. Nuseibeh, "Security requirements engineering: A framework for representation and analysis," *IEEE Transactions on Software Engineering*, vol. 34, no. 1, pp. 133–153, 2008.

[2] M. Felderer, M. Buchler, M. Johns, A. D. Brucker, R. Breu, and A. Pretschner, "Security testing: A survey," *Advances in Computers*, vol. 101, pp. 1–51, 2016.

[3] P. X. Mai, A. Goknil, L. K. Shar, F. Pastore, L. C. Briand, and S. Shaame, "Modeling security and privacy requirements: a use case-driven approach," *Information and Software Technology*, vol. 100, pp. 165–182, 2018.

[4] P. X. Mai, F. Pastore, A. Goknil, and L. C. Briand, "A natural language programming approach for requirements-based security testing," in *Proceedings of 29th IEEE International Symposium on Software Reliability Engineering (ISSRE'18)*, 2018, pp. 58–69, note: the paper reports on E2 vulnerabilities targeted in Section X. They concern OTG-AUTHN-001, OTG-AUTHN-004, OTG-AUTHN-010, OTG-BUSLOGIC-005.

[5] ——, "Mcp: a security testing tool driven by requirements," in *ICSE (Companion Volume)'19*, 2019, pp. 55–58.

[6] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[7] M. Staats, M. W. Whalen, and M. P. Heimdahl, "Programs, tests, and oracles: the foundations of testing revisited," in *ICSE'11*, 2011, pp. 391–400.

[8] M. Pezze and C. Zhang, "Automated test oracles: A survey," *Advances in Computers*, vol. 95, pp. 1–48, 2014.

[9] M. Meucci and A. Muller, "OWASP Testing Guide v4," https://www.owasp.org/images/1/19/OTGv4.pdf.

[10] G. Rosen, "Facebook Security Update on 'View As' Vulnerability," https://newsroom.fb.com/news/2018/09/security-update/.

[11] D. Deahl, "Another Facebook Vulnerability," https://bit.ly/3gtPo80.

[12] E. 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.

[13] J. 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.

[14] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *USENIX Security'13*, 2013, pp. 49–64.

[15] S. Ognawala, M. Ochoa, A. Pretschner, and T. Limmer, "MACKE: Compositional analysis of low-level vulnerabilities with symbolic execution," in *ASE'16*, 2016, pp. 780–785.

[16] T. Y. Chen, S.-C. Cheung, and S.-M. Yiu, "Metamorphic testing: a new approach for generating next test cases," The Hong Kong University of Science and Technology, Tech. Rep., 1998.

[17] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen, "How effectively does metamorphic testing alleviate the oracle problem?" *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 4–22, 2014.

[18] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortes, "A survey on metamorphic testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[19] J. Mayer and R. Guderlei, "On random testing of image processing applications," in *QSIC'06*, 2006, pp. 85–92.

[20] R. Guderlei and J. Mayer, "Towards automatic testing of imaging software by means of random and metamorphic testing," *International Journal of Software Engineering and Knowledge Engineering*, vol. 17, no. 6, pp. 757–781, 2007.

[21] R. Just and F. Schweiggert, "Evaluating testing strategies for imaging software by means of mutation analysis," in *ICSTW'09*, 2009, pp. 205–209.

[22] F.-C. Kuo, S. Liu, and T. Y. Chen, "Testing a binary space partitioning algorithm with metamorphic testing," in *SAC'11*, 2011, pp. 1482–1489.

[23] W. K. Chan, S. C. Cheung, and K. R. Leung, "A metamorphic testing approach for online testing of service-oriented software applications," *International Journal of Web Services Research*, vol. 4, no. 2, pp. 61–81, 2007.

[24] C.-a. Sun, G. Wang, B. Mu, H. Liu, Z. Wang, and T. Y. Chen, "Metamorphic testing for web services: Framework and a case study," in *ICWS'11*, 2011, pp. 283–290.

[25] Z. Q. Zhou, S. Zhang, M. Hagenbuchner, T. Tse, F.-C. Kuo, and T. Y. Chen, "Automated functional testing of online search services," *Software: Testing, Verification and Reliability*, vol. 22, no. 4, pp. 221–243, 2012.

[26] T. Tse and S. S. Yau, "Testing context-sensitive middleware-based software applications," in *COMPSAC'04*, 2004, pp. 458–466.

[27] W. K. Chan, T. Y. Chen, S. C. Cheung, T. Tse, and Z. Zhang, "Towards the testing of power-aware software applications for wireless sensor networks," in *ADA Europe'07*, 2007, pp. 84–99.

[28] F.-C. Kuo, T. Y. Chen, and W. K. Tam, "Testing embedded software by metamorphic testing: A wireless metering system case study," in *LCN'11*, 2011, pp. 291–294.

[29] M. Jiang, T. Y. Chen, F.-C. Kuo, and Z. Ding, "Testing central processing unit scheduling algorithms using metamorphic testing," in *ICSESS'13*, 2013, pp. 530–536.

[30] T. Y. Chen, F. Kuo, W. Ma, W. Susilo, D. Towey, J. Voas, and Z. Q. Zhou, "Metamorphic testing for cybersecurity," *Computer*, vol. 49, no. 6, pp. 48–55, June 2016.

[31] Synopsys Inc., "Description of the openssl heartbleed vulnerability." http://heartbleed.com/.

[32] "Eclipse IDE, https://www.eclipse.org/ide/."

[33] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling Ajax-based web applications through dynamic analysis of user interface state changes," *ACM Transactions on the Web (TWEB)*, vol. 6, no. 1, pp. 3:1–3:30, 2012.

[34] "JUnit, https://junit.org/."

[35] "Open Web Application Security Project." https://www.owasp.org/.

[36] "MITRE common weaknesses enumeration project, which provide a taxonomy of vulnerability types." https://cwe.mitre.org.

[37] Eclipse Foundation, "Jenkins ci/cd server." https://jenkins.io/.

[38] "Joomla, https://www.joomla.org/."

[39] P. X. Mai, F. Pastore, A. Goknil, and L. C. Briand, "Metamorphic security testing for web systems," in *ICST'20*, 2020, pp. 186–197.

[40] P. X. Mai, A. Goknil, F. Pastore, and L. C. Briand, "SMRL: a metamorphic security testing tool for web systems," in *ICSE (Companion Volume)'20*, 2020, pp. 9–12.

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

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

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

[44] P. X. Mai, "CVE-2020-2162: Stored XSS vulnerability in file parameters," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-2162, 2020.

[45] N. B. Chaleshtari, P. X. Mai, F. Pastore, A. Goknil, and L. C. Briand, "MST library." https://github.com/MetamorphicSecurityTesting/MST.

[46] P. X. Mai, F. Pastore, A. Goknil, and L. C. Briand, "SMRL editor." https://github.com/SNTSVV/SMRL_EclipsePlugin/.

[47] N. B. Chaleshtari, P. X. Mai, F. Pastore, A. Goknil, and L. C. Briand, "Catalog of metamorphic relations for MST." https://github.com/MetamorphicSecurityTesting/CWE.

[48] N. B. Chaleshtari, F. Pastore, A. Goknil, and L. C. Briand, "Replicability package," 2023, https://doi.org/10.5281/zenodo.7702754.

[49] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou, "Metamorphic testing: A review of challanges and opportunities," *ACM Computing Surveys*, vol. 51, no. 1, 2018.

[50] Microsoft Corp., "Silverlight plug-ins and development tools." https://www.microsoft.com/silverlight/.

[51] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, and M. Hanus, "Xbase: Implementing domain-specific languages for java," *ACM SIGPLAN Notices - GPCE '12*, vol. 48, no. 3, pp. 112–121, 2012.

[52] "Xtext, https://www.eclipse.org/Xtext/."

[53] N. B. Chaleshtari, P. X. Mai, F. Pastore, A. Goknil, and L. C. Briand, "SMRL editor executable, catalog of MRs, MT framework, experimental data." https://sntsvv.github.io/SMRL/.

[54] A. Mesbah, A. Van Deursen, and S. Lenselink, "Crawling ajax-based web applications through dynamic analysis of user interface state changes," *ACM Transactions on the Web*, vol. 6, no. 1, p. 3, 2012.

[55] A. Mesbah, E. Bozdag, and A. Van Deursen, "Crawling ajax by inferring user interface state changes," in *ICWE'08*, 2008, pp. 122–134.

[56] V. I. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady*, vol. 10, Feb. 1966.

[57] "Selenium Web Testing Framework, https://www.seleniumhq.org/."

[58] "ASM bytecode manipulation framework." https://asm.ow2.io/.

[59] "CWE - Common Weakness Enumeration," https://cwe.mitre.org.

[60] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "The fuzzing book," in *The Fuzzing Book*. Saarland University, 2019, retrieved 2019-09-09 16:42:54+02:00. [Online]. Available: https://www.fuzzingbook.org/

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

[62] J. M. Voas and K. W. Miller, "Software testability: The new verification," *IEEE software*, vol. 12, no. 3, pp. 17–28, 1995.

[63] "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.

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

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

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

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

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

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

[70] J. C. Santos, A. Peruma, M. Mirakhorli, M. Galstery, J. V. Vidal, and A. Sejfia, "Understanding software vulnerabilities related to architectural security tactics: An empirical investigation of chromium, php and thunderbird," in *ICSA'17*, 2017, pp. 69–78.

[71] J. C. Santos, K. Tarrit, and M. Mirakhorli, "A catalog of security architecture weaknesses," in *ICSAW'17*, 2017, pp. 220–223.

[72] N. M. 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," LOCKHEED MARTIN INC FORT WORTH TX, Tech. Rep., 2011, https://apps.dtic.mil/docs/citations/ADB381215.

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

[74] "OWASP Top 10 Mobile Security Risks," https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10.

[75] OWASP, "WSTG-ATHZ-02: Testing for Bypassing Authorization Schema." https://bit.ly/2Y6SloC.

[76] ——, "OTG-INFO-010: Mapping application architecture." https://www.owasp.org/index.php/Map_Application_Architecture_(OTG-INFO-010).

[77] ——, "OTG-BUSLOGIC-006: Testing for the circumvention of workflows." https://www.owasp.org/index.php/Testing_for_the_Circumven\-tion_of_Work_Flows_(OTG-BUSLOGIC-006).

[78] ——, "OTG-INPVAL-014: Testing for Buffer Overflow." https://www.owasp.org/index.php/Testing_for_Buffer_Overflow_(OTG-INPVAL-014).

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

[80] OWASP, "OTG-AUTHN-002: Testing for default credentials." https://www.owasp.org/index.php/Testing_for_default_credentials_(OTG-AUTHN-002).

[81] Portswigger, "Burp suite." https://portswigger.net/burp.

[82] ——, "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.

[83] ——, "Burp suite scanning (crawling) feature." https://portswigger.net/burp/documentation/desktop/scanning.

[84] R. S. Liverani, "Integration of burp suite and crawljax." https://github.com/portswigger/burp-csj.

[85] OWASP, "WSTG-ATHZ-01: Testing Directory Traversal File Include." https://bit.ly/3l1sRTl.

[86] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2019.

[87] OWASP, "WSTG-SESS-03: Testing for Session Fixation." https://bit.ly/34myBkN.

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

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

[90] S. Elder, N. Zahan, R. Shu, M. Metro, V. Kozarev, T. Menzies, and L. Williams, "Do I really need all this work to find vulnerabilities?" *Empirical Software Engineering*, vol. 27, no. 6, p. 154, 2022. [Online]. Available: https://doi.org/10.1007/s10664-022-10179-6

[91] OWASP, "OWASP Zed Attack Proxy," https://www.zaproxy.org.

[92] SonarSource, "Sonarqube: Code Quality and Code Security toolset," https://www.sonarqube.org/.

[93] A. Austin, C. Holmgreen, and L. Williams, "A comparison of the efficiency and effectiveness of vulnerability discovery techniques," *Information and Software Technology*, vol. 55, no. 7, pp. 1279–1288, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584912002339

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

[95] V. Garousi, M. Felderer, and F. N. Kılıçaslan, "A survey on software testability," *Information and Software Technology*, vol. 108, pp. 35–64, 2019.

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

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

[98] Eclipse Foundation, "Jetty application server." https://www.eclipse.org/jetty/.

[99] Oracle corp., "MYSQL RDBMS engine," 2022, https://www.mysql.com/.

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

[101] MITRE Corporation, "Common vulnerabilities and exposures." https://cve.mitre.org/cve/.

[102] MITRE, "CVE-2018-1999003, concerns OTG-AUTHZ-002," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999003.

[103] ——, "CVE-2018-1999004, concerns OTG-AUTHZ-002," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999004.

[104] ——, "CVE-2018-1999006, concerns OTG-AUTHZ-002," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999006.

[105] ——, "CVE-2018-1000406, concerns OTG-AUTHN-001," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1000406.

[106] ——, "CVE-2018-1999046, concerns OTG-AUTHZ-002," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999046.

[107] ——, "CVE-2018-1000409, concerns OTG-SESS-003," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1000409.

[108] ——, "CWE-521," https://cwe.mitre.org/data/definitions/521.html.
[109] ——, "CWE-262," https://cwe.mitre.org/data/definitions/262.html.
[110] ——, "CVE-2020-2162, concerns OTG-INPVAL-003," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-2162.
[111] ——, "CVE-2018-1999047, concerns OTG-AUTHZ-002," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999047.
[112] ——, "CVE-2018-1999045, concerns OTG-AUTHZ-002," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999045.
[113] ——, "CVE-2018-17857," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-17857.
[114] ——, "CVE-2018-11327," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-11327.
[115] D. Lane, *Online Statistics Education: A Multimedia Course of Study*. Rice University, 2003. [Online]. Available: http://onlinestatbook.com/
[116] R. Wood, "Damn vulnerable web application (dvwa)," 2022, https://dvwa.co.uk/.
[117] P. Ralph and E. 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*, vol. Part F1377. New York, NY, USA: ACM, jun 2018, pp. 13–23. [Online]. Available: https://dl.acm.org/doi/10.1145/3210459.3210461
[118] "EDLAH2: Active and Assisted Living Programme," http://www.edlah2.eu/.
[119] M. Felderer, P. Zech, R. Breu, M. Büchler, and A. Pretschner, "Model-based security testing: A taxonomy and systematic classification," *Software: Testing, Verification and Reliability*, vol. 26, no. 2, pp. 119–148, 2016.
[120] G. Tian-yang, S. Yin-Sheng, and F. You-yuan, "Research on software security testing," *World Academy of Science, Engineering and Technology*, vol. 70, no. 69, pp. 647–651, 2010.
[121] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "Finding software vulnerabilities by smart fuzzing," in *ICST'11*, 2011, pp. 427–430.
[122] A. Takanen, J. D. Demott, C. Miller, and A. Kettunen, *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2018.
[123] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic, "SecuBat: A web vulnerability scanner," in *WWW'06*, 2006, pp. 247–246.
[124] M. Martin and M. S. Lam, "Automatic generation of XSS and SQL injection attacks with goal-directed model checking," in *USENIX Security'08*, 2008, pp. 31–43.
[125] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the art: Automated black-box web application vulnerability testing," in *SP'10*, 2010, pp. 332–345.
[126] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan, "Automated testing for sql injection vulnerabilities: An input mutation approach," in *ISSTA'14*, 2014, pp. 259–269.
[127] M. Salas and E. Martins, "Security testing methodology for vulnerabilities detection of XSS in web services and WS-security," *ENTCS*, pp. 133–154, 2014.
[128] O. Tripp, O. Weisman, and L. Guy, "Finding your way in the testing jungle: A learning approach to web security testing," in *ISSTA'13*, 2013, pp. 347–357.
[129] D. Appelt, N. Alshahwan, and L. Briand, "Assessing the impact of firewalls and database proxies on sql injection testing," in *FITTEST'13*, 2013, pp. 32–47.
[130] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai, "Web application security assessment by fault injection and behavior monitoring," in *WWW'03*, 2003, pp. 148–159.
[131] Y. Le Traon, T. Mouelhi, and B. Baudry, "Testing security policies: Going beyond functional testing," in *ISSRE'07*, 2007, pp. 93–102.
[132] T. Mouelhi, F. Fleurey, B. Baudry, and Y. Le Traon, "A model-based framework for security policy specification, deployment and testing," in *MODELS'08*, 2008, pp. 537–552.
[133] T. Mouelhi, Y. Le Traon, and B. Baudry, "Transforming and selecting functional test cases for security policy testing," in *ICST'09*, 2009, pp. 171–180.
[134] E. Martin and T. Xie, "Automated test generation for access control policies via change-impact analysis," in *SESS'07*, 2007.
[135] ——, "A fault model and mutation testing of access control policies," in *WWW'07*, 2007, pp. 667–676.
[136] G. Wimmel and J. Jürjens, "Specification-based test generation for security-critical systems using mutations," in *ICFEM'02*, 2002, pp. 471–482.
[137] M. Masood, A. Ghafoor, and A. Mathur, "Conformance testing of temporal role-based access control systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 2, pp. 144–158, 2010.
[138] A. Bertolino, S. Daoudagh, F. Lonetti, E. Marchetti, F. Martinelli, and P. Mori, "Testing of PolPA authorization systems," in *AST'12*, 2012, pp. 8–14.
[139] A. Blome, M. Ochoa, K. Li, M. Peroli, and M. T. Dashti, "Vera: A flexible model-based vulnerability testing tool," in *ICST'13*, 2013, pp. 471–478.
[140] K. He, Z. Feng, and X. Li, "An attack scenario based approach for software security testing at design stage," in *ISCSCT'08*, 2008, pp. 782–787.
[141] A. Marback, H. Do, K. He, S. Kondamarri, and D. Xu, "A threat model-based approach to security testing," *Software: Practice and Experience*, vol. 43, no. 2, pp. 241–258, 2013.
[142] J. Jürjens, "Model-based security testing using UMLsec: A case study," *Electronic Notes in Theoretical Computer Science*, vol. 220, no. 1, pp. 93–104, 2008.
[143] D. Xu and K. E. Nygard, "Threat-driven modeling and verification of secure software using aspect-oriented petri nets," *IEEE Transactions on Software Engineering*, vol. 32, no. 4, pp. 265–278, 2006.
[144] F. Lebeau, B. Legeard, F. Peureux, and A. Vernotte, "Model-based vulnerability testing for web applications," in *ICSTW'13*, 2013, pp. 445–452.
[145] J. Whittle, D. Wijesekera, and M. Hartong, "Executable misuse cases for modeling security concerns," in *ICSE'08*, 2008, pp. 121–130.
[146] D. Xu, M. Tu, M. Sanford, L. Thomas, D. Woodraska, and W. Xu, "Automated security test generation with formal threat models," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 4, pp. 526–540, 2012.
[147] D. Xu, W. Xu, M. Kent, L. Thomas, and L. Wang, "An automated test generation technique for software quality assurance," *IEEE Transactions on Reliability*, vol. 64, no. 1, pp. 247–268, 2015.
[148] D. Xu, L. Thomas, M. Kent, T. Mouelhi, and Y. Le Traon, "A model-based approach to automated testing of access control policies," in *SACMAT'12*, 2012, pp. 209–218.
[149] T. Jameel, M. Lin, and L. Chao, "Test oracles based on metamorphic relations for image processing applications," in *SNPD'15*, 2015, pp. 1–6.
[150] W. Chan, J. C. Ho, and T. Tse, "Piping classification to metamorphic testing: An empirical study towards better effectiveness for the identification of failures in mesh simplification programs," in *COMPSAC'07*, vol. 1, 2007, pp. 397–404.
[151] W. K. Chan, J. C. Ho, and T. Tse, "Finding failures from passed test cases: Improving the pattern classification approach to the testing of mesh simplification programs," *Software Testing, Verification and Reliability*, vol. 20, no. 2, pp. 89–120, 2010.
[152] T. Y. Chen, F.-C. Kuo, H. Liu, and S. Wang, "Conformance testing of network simulators based on metamorphic testing technique," in *FORTE'09*, 2009, pp. 243–248.
[153] J. Ding, T. Wu, D. Wu, J. Q. Lu, and X.-H. Hu, "Metamorphic testing of a monte carlo modeling program," in *AST'11*, 2011, pp. 1–7.
[154] C. Murphy, M. S. Raunak, A. King, S. Chen, C. Imbriano, G. Kaiser, I. Lee, O. Sokolsky, L. Clarke, and L. Osterweil, "On effective testing of health care simulation software," in *SEHC'11*, 2011, pp. 40–47.
[155] K. Sim, W. Pao, and C. Lin, "Metamorphic testing using geometric interrogation technique and its application," *ECTI Transactions on Computer and Information Technology (ECTI-CIT)*, vol. 1, no. 2, pp. 91–95, 2005.
[156] Q. Tao, W. Wu, C. Zhao, and W. Shen, "An automatic testing approach for compiler based on metamorphic testing technique," in *APSEC'10*, 2010, pp. 270–279.
[157] Y. Yao, S. Huang, and M. Ji, "Research on metamorphic testing for oracle problem of integer bugs," in *AISC'12*, 2012, pp. 95–100.
[158] S. Segura, R. M. Hierons, D. Benavides, and A. Ruiz-Cortés, "Automated metamorphic testing on the analyses of feature models," *Information and Software Technology*, vol. 53, no. 3, pp. 245–258, 2011.
[159] ——, "Automated test data generation on the analyses of feature models: A metamorphic testing approach," in *ICST'10*, 2010, pp. 35–44.

[160] F.-C. Kuo, Z. Q. Zhou, J. Ma, and G. Zhang, "Metamorphic testing of decision support systems: a case study," *IET software*, vol. 4, no. 4, pp. 294–301, 2010.
[161] T. Y. Chen, J. W. Ho, H. Liu, and X. Xie, "An innovative approach for testing bioinformatics programs using metamorphic testing," *BMC bioinformatics*, vol. 10, no. 1, p. 24, 2009.
[162] L. L. Pullum and O. Ozmen, "Early results from metamorphic testing of epidemiological models," in *BioMedCom'12*, 2012, pp. 62–67.
[163] T. Y. Chen, J. Feng, and T. Tse, "Metamorphic testing of programs on partial differential equations: a case study," in *COMPSAC'02*, 2002, pp. 327–333.
[164] C. Aruna and R. S. R. Prasad, "Metamorphic relations to improve the test accuracy of multi precision arithmetic software applications," in *ICACCI'14*, 2014, pp. 2244–2248.
[165] X. Xie, J. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Application of metamorphic testing to supervised classifiers," in *QSIC'09*, 2009, pp. 135–144.
[166] C. Murphy, G. E. Kaiser, and L. Hu, "Properties of machine learning applications for use in metamorphic testing," Columbia University, Tech. Rep., 2008.
[167] S. Segura, J. A. Parejo, J. Troya, and A. Ruiz-Cortés, "Metamorphic testing of RESTful web APIs," *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1083–1099, 2017.
[168] S. Segura, A. Durán, J. Troya, and A. R. Cortés, "A template-based approach to describing metamorphic relations," in *MET'17*, 2017, pp. 3–9.
[169] S. Segura, A. Durán, J. Troya, and A. Ruiz-Cortés, "Metamorphic relation template v1. 0," *Applied Software Engineering Research Group, Tech. Rep. ISA-17-TR-01*, 2017.
[170] C.-a. Sun, Z. Wang, and G. Wang, "A property-based testing framework for encryption programs," *Frontiers of Computer Science*, vol. 8, no. 3, pp. 478–489, 2014.
[171] Q.-H. Luu, M. F. Lau, S. P. Ng, and T. Y. Chen, "Testing multiple linear regression systems with metamorphic testing," *Journal of Systems and Software*, vol. 182, p. 111062, 2021.
[172] A. Lascu, A. F. Donaldson, T. Grosser, and T. Hoefler, "Metamorphic fuzzing of c++ libraries," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2022, pp. 35–46.
[173] J. Ayerdi, V. Terragni, A. Arrieta, P. Tonella, G. Sagardui, and M. Arratibel, "Generating metamorphic relations for cyber-physical systems with genetic programming: an industrial case study," in *ESEC/FSE'21*, 2021, pp. 1264–1274.
[174] L. Xu, D. Towey, A. P. French, S. Benford, Z. Q. Zhou, and T. Y. Chen, "Using metamorphic relations to verify and enhance artcode classification," *Journal of Systems and Software*, vol. 182, p. 111060, 2021.
[175] S. Raghavan and H. Garcia-Molina, "Crawling the hidden web," in *VLDB'01*, 2000, pp. 129–138.
[176] M. Felderer, B. Agreiter, P. Zech, and R. Breu, "A classification for model-based security testing," in *VALID'11*, 2011, pp. 109–114.
[177] J. Jürjens, "UMLsec: Extending UML for secure systems development," in *UML'02*, 2002, pp. 412–425.
[178] ——, "Sound methods and effective tools for model-based security engineering with UML," in *ICSE'05*, 2005, pp. 322–331.
[179] ——, *Secure Systems Development with UML*. Springer Science & Business Media, 2005.
[180] E. Martin, T. Xie, and T. Yu, "Defining and measuring policy coverage in testing access control policies," in *ICICS'06*, 2006, pp. 139–158.
[181] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 216–226, 2014.
[182] S. Segura, A. Durán, A. B. Sánchez, D. L. Berre, E. Lonca, and A. Ruiz-Cortés, "Automated metamorphic testing of variability analysis tools," *Software Testing, Verification and Reliability*, vol. 25, no. 2, pp. 138–163, 2015.
[183] A. Ramanathan, C. A. Steed, and L. L. Pullum, "Verification of compartmental epidemiological models using metamorphic testing, model checking and visual analytics," in *BioMedCom'12*, 2012, pp. 68–73.
[184] H. Zhu, "Jfuzz: A tool for automated java unit testing based on data mutation and metamorphic testing methods," in *TSA'15*, 2015, pp. 8–15.
[185] C. Murphy, K. Shen, and G. Kaiser, "Using JML Runtime Assertion Checking to Automate Metamorphic Testing in Applications without Test Oracles," in *ICST'09*, 2009, pp. 436–445.
[186] C. Murphy, G. E. Kaiser, and L. Hu, "Properties of machine learning applications for use in metamorphic testing," Columbia University, Tech. Rep., 2008. [Online]. Available: https://doi.org/10.7916/D8XK8PFD
[187] Z. Q. Zhou, L. Sun, T. Y. Chen, and D. Towey, "Metamorphic relations for enhancing system understanding and use," *IEEE Transactions on Software Engineering*, vol. 46, no. 10, pp. 1120–1154, 2018.
[188] S. Segura, A. Durán, J. Troya, and A. Ruiz-Cortés, "Metamorphic relation patterns for query-based systems," in *2019 IEEE/ACM 4th International Workshop on Metamorphic Testing (MET)*, 2019, pp. 24–31.

Nazanin Bayati Chaleshtari is a Ph.D. student at the School of EECS at the University of Ottawa and a member of the Nanda Lab. She received several academic awards, including a Ph.D. admission scholarship, an international doctoral scholarship from the University of Ottawa, and an honourable award for being an outstanding student during her master's degree at the Iran University of Science and Technology. She was also ranked the best student among all computer engineering students at the Iran University of Science and Technology in 2019. Her research interests include automated software testing concerning security testing, applied data science and empirical software engineering.

Fabrizio Pastore is Chief Scientist II at the Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg. He obtained his PhD in Computer Science in 2010 from the University of Milano - Bicocca.

His research interests concern automated software testing, including security testing and testing of AI-based systems; his work relies on the integrated analysis of different types of artefacts (e.g., requirements, models, source code, and execution traces). He is active in several industry partnerships and national, ESA, and EU-funded research projects.

Arda Goknil received his Ph.D. degree in computer science from the University of Twente, in the Netherlands, in 2011. He is a senior research scientist at SINTEF, Norway. He was a research associate at the Interdisciplinary Centre for Security, Reliability, and Trust (SnT) at the University of Luxembourg. His research concerns AI Engineering, Model-Driven Engineering, Software Testing, Software Security, Intermittent Computing, Product Line Engineering, and Requirements Engineering. He is active on EU-funded and national research projects with several academic and industry partners.

Lionel C. Briand is professor of software engineering and has shared appointments between (1) School of Electrical Engineering and Computer Science, University of Ottawa, Canada and (2) The SnT centre for Security, Reliability, and Trust, University of Luxembourg. He is the head of the SVV department at the SnT Centre and a Canada Research Chair in Intelligent Software Dependability and Compliance (Tier 1).

He has conducted applied research in collaboration with industry for more than 25 years, including projects in the automotive, aerospace, manufacturing, financial, and energy domains. In 2016, he received an ERC Advanced grant, the most prestigious European individual research award. He was elevated to the grades of IEEE and ACM fellow, granted the ACM SIGSOFT Outstanding Research Award (2022), the IEEE Computer Society Harlan Mills award (2012), and the IEEE Reliability Society Engineer-of-the-year award (2013) for his work on software verification and testing. His research interests include: Testing and verification, trustworthy AI, search-based software engineering, model-driven development, requirements engineering, and empirical software engineering.