SMRL: A Metamorphic Security Testing Tool for Web Systems

Phu X. Mai
SnT, University of Luxembourg

Arda Goknil
SnT, University of Luxembourg

Fabrizio Pastore
SnT, University of Luxembourg

Lionel C. Briand
SnT, University of Luxembourg
University of Ottawa

ABSTRACT

We present a metamorphic testing tool that alleviates the oracle problem in security testing. The tool enables engineers to specify metamorphic relations that capture security properties of Web systems. It automatically tests Web systems to detect vulnerabilities based on those relations. We provide a domain-specific language accompanied by an Eclipse editor to facilitate the specification of metamorphic relations. The tool automatically collects the input data and transforms the metamorphic relations into executable Java code in order to automatically perform security testing based on the collected data. The tool has been successfully evaluated on a commercial system and a leading open source system (Jenkins). Demo video: https://youtu.be/9kx6u9LsGxs.

CCS CONCEPTS

• **Security and privacy → Software and application security**; • **Software and its engineering → Software verification and validation**.

ACM Reference Format:
Phu X. Mai, Arda Goknil, Fabrizio Pastore, and Lionel C. Briand. 2020. SMRL: A Metamorphic Security Testing Tool for Web Systems. In *42nd International Conference on Software Engineering Companion (ICSE '20 Companion), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3377812.3382152

1 INTRODUCTION

Security testing aims at verifying that the software meets its security properties [14–16]. In modern Web systems, this often entails the verification of the outputs generated when exercising the system against a very large set of inputs [17]. For example, authorization problems might be discovered only by accessing all the resources of the system with different user profiles and by verifying if the system grants access only to authorized users. Full automation will thus lower costs and increase the effectiveness of security testing. Unfortunately, although Web crawlers provide means to automatically exercise Web systems, there is a lack of solutions to automatically verify the correctness of outputs. Therefore, security testing suffers from the oracle problem [4], which refers to situations where it is extremely difficult to determine the correct output for a given test input (e.g., the correct response to be received after a specific HTTP GET request). State-of-the-art approaches do not address the oracle problem and assume the availability of an implicit oracle [4, 17].

Metamorphic Testing (MT) has proved, in some contexts, to be very effective to alleviate the oracle problem [8, 13]. *MT is based on the idea that it may be simpler to reason about relations between outputs of multiple test executions, called metamorphic relations (MRs), than it is to specify its input-output behaviour* [22]. In MT, system properties are captured as MRs that are used to automatically transform an initial set of test inputs (source inputs) into follow-up test inputs. If the system outputs for the initial and follow-up test inputs violate the MR, it is concluded that the system is faulty.

There is considerable research devoted to developing MT approaches (e.g., [7, 11]). However, all these approaches focus on testing functional requirements, not potential security threats within the context of security testing. Although MT is automatable, very few MT approaches provide proper tool support [22]. This is also a significant obstacle for tailoring the current MT approaches in the context of security testing.

In this paper, we present the SMRL (Security Metamorphic Relation Language) tool, which enables engineers to specify MRs that capture security properties of Web systems. Our tool automatically detects vulnerabilities (i.e., property violations) based on those relations. It is built on top of the following novel contributions [17]: (1) a Domain-Specific Language (DSL) for specifying MRs for software security, (2) a set of system-agnostic, MRs targeting well-known security vulnerabilities of Web systems [19], (3) a framework automatically collecting the input data required to perform MT, and (4) a testing framework automatically performing security testing based on the MRs and the collected data. To facilitate the specification of MRs in our DSL, we provide an editor implemented as a plug-in for Eclipse IDE.

In the remaining sections, we outline the tool and highlight the findings from our evaluation of the tool over two case studies.

2 RELATED WORK

Several security testing approaches have been proposed relying on an implicit test oracle that only deals with simple abnormal system behavior such as unexpected system termination (e.g., buffer overflows, memory leaks, and denial of service [5, 21]). What is abnormal in one system might be considered normal in a different context [4]. Our tool complements these approaches for cases in which it is impractical to define an exact oracle for each test case.

A number of solutions and tools provide support for MT in various domains such as computer graphics (e.g., [11]) and embedded systems (e.g., [7]). None of these approaches facilitate the specification of MRs capturing security properties or targeting vulnerabilities. MT is highly automatable. However, there is a lack of tool support that enables engineers to write system-level MRs [22]. The tools require that relations be defined as JUnit methods [23] or pre-/post-conditions [20]. This is an obstacle to the adoption of the current MT approaches in security testing. To the best of our knowledge, the SMRL tool is the first MT tool addressing the oracle problem in the context of security testing.

3 OVERVIEW OF THE SMRL TOOL

The SMRL tool supports our approach for security testing based on MRs, described in our recent research paper [17].

Fig. 1 provides an overview of our tool. It consists of (1) an Eclipse plugin, which provides the Editor for our DSL and automatically generates Java code from MRs, (2) a library (i.e., `SMRL.jar`), which implements our MT algorithm and provides utility functions supporting the writing of MRs, (3) an extension of the Crawljax Web crawler [18] to collect source inputs for MT. The SMRL tool relies on JUnit to automatically execute MT within Eclipse through our library. The Eclipse workspace is used to store all the data, which includes MRs and source inputs.

Fig. 2 presents an overview of our approach. In Step 1, the engineer specifies MRs with our Eclipse editor. We derived a catalog of MRs from the OWASP testing guidelines [19]. The engineer can also select, from this catalog, the MRs for the system under test. Selection is performed by specifying the MRs to be executed in a JUnit test, i.e., in the same way as for JUnit test suites. In Step 2, the relations are automatically transformed into Java code.

In Step 3, Crawljax is used to automatically collects the source inputs for MT (e.g., the URLs that can be visited by an anonymous user). In Step 4, testing is automatically performed based on the executable relations and the collected data. In the rest of the section, we elaborate each step in Fig. 2.

3.1 Specification of Metamorphic Relations

As a first step, the engineer specifies MRs. To enable specifying new relations, we provide the Security Metamorphic Relation Language (SMRL) as an extension of Xbase [10]. Xbase is a statically typed expression language for Java, implemented in Xtext. Its specifications can be translated to Java code. Xbase is extended by SMRL with data representation functions, boolean operators and Web-specific functions. New Java APIs can be defined to extend these functions.

Fig. 3 presents an MR in our SMRL editor. In an SMRL specification, we first import the SMRL operators and functions (Line 1 in Fig. 3). We then define a package for MRs (Line 4). SMRL supports 18 data representation functions to represent system inputs and outputs. We represent data with some keywords followed by an index number identifying data items (e.g., `Input(1)`, `Input(2)` and `User(2)` in Line 12). `Input(1)` returns the first input sequence, while `User(2)` returns the second user of the system. `Output` is a data function which executes the actions in an input sequence (e.g., requests a sequence of URLs) and returns the sequence of corresponding outputs (Lines 11, 14 and 15).

SMRL supports boolean operators `IMPLIES`, `AND`, `OR`, `TRUE`, `FALSE`, `NOT` and `EQUAL`. `EQUAL` does not necessarily evaluate the equality of two arguments. It can also be used to define a follow-up input. For instance, in Fig. 3, the follow-up input `Input(2)` is defined by `EQUAL` as a modified copy of `Input(1)` by assigning the second parameter to the first parameter. `EQUAL` acts as an equality operator only when its first parameter refers to an input that has already been used in previous expressions of the MR .

We use four Web-specific functions (`cannotReachThroughGUI`, `isSupervisorOf`, `isError`, and `changeCredentials`) in Fig. 3. In the MR in Fig. 3, the same sequence of actions should provide different outputs for two different users under a condition. According to the condition, the user should not be allowed to access URLs that are not provided through a GUI. `cannotReachThroughGUI` checks if the URL of the current action cannot be reached from the GUI (Line 9). `isSupervisorOf` checks if `User(2)` is a supervisor of `User(1)` (Line 10). `isError` uses configurable regular expressions to check if an output page contains an error message (Line 11). `changeCredentials` defines the follow-up input (Line 12). It creates a copy of the input sequence using different credentials.

3.2 Transformation of MRs into Java

Our tool automatically transforms SMRL specifications into Java code. It employs the SMRL compiler, an Xbase compiler extended with transformation rules in Xtend (a language provided with Xbase [6]). Each MR is transformed into a Java class with the name of the relation and its package. The generated classes extend the class `MR` and implement its method `mr`.

The metamorphic expressions in the relation are executed by the method `mr`. The method returns `true` if the relation holds and `false` otherwise. The SMRL compiler creates a set of nested `IF` conditions for each boolean operator in the relation. For example, for `IMPLIES`, `mr` returns `false` when the first parameter is true and the second one is false. For the case in which the relation holds, `mr` returns `true` at the end. Fig. 4 shows the Java code generated from the relation in Fig. 3.

3.3 Data Collection

The SMRL tool automatically derives source inputs for MT. To do so, we extended the Crawljax Web crawler [18]. Crawljax explores the user interface of a Web system by requesting URLs in HTML anchors or by entering text in HTML forms. Its output is graphs having nodes representing the system states reached through the user interface and edges capturing the actions performed to reach these states. System states are detected based on the content of the displayed page. To reduce the number of system states, the Crawljax extension distinguishes states by using the edit distance [12]. Our tool caches HTML pages associated to states identified by Crawljax. It computes the edit distance between the new loaded page and the cached pages. It is assumed that two pages belong to the same state if the distance is below a given threshold (5% of the page length); otherwise, a new state is added to the graph. To crawl the system under test, our tool requires only its URL and a list of credentials. It stops crawling the system under test when there is no more state for a certain number of new pages or a timeout is reached.

Our tool automatically extracts source inputs from the Crawljax graphs to be later queried by the SMRL functions during test execution (see Section 3.4). Fig. 5 shows two example Crawljax graphs and the source inputs extracted from the graphs. For example, the

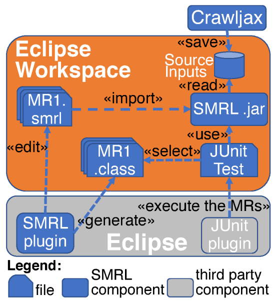


Figure 1: Components of the SMRL tool.

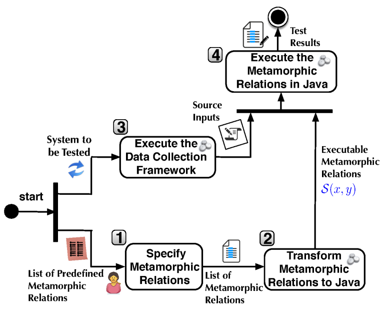
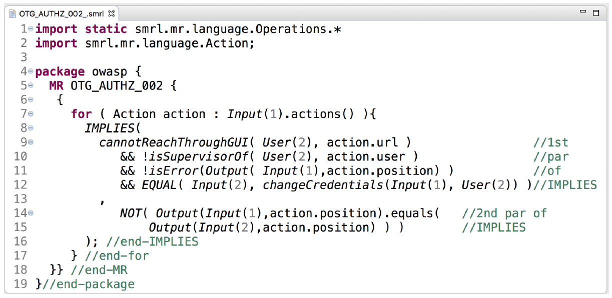

Test Results
4
Execute the Metamorphic Relations in Java
Source Inputs
3
System to be Tested
Execute the Data Collection Framework
Executable Metamorphic Relations
$\mathcal{S}(x, y)$
start
1
Specify Metamorphic Relations
List of Predefined Metamorphic Relations
List of Metamorphic Relations
2
Transform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

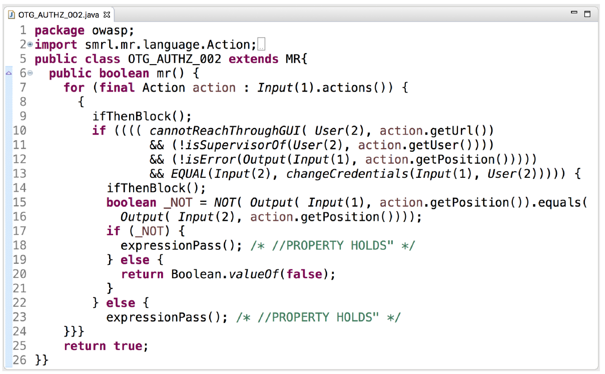
Figure 2: Overview of the approach.



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

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

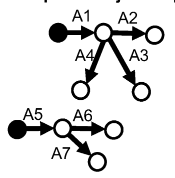
Figure 3: An MR for the Bypass Authorization Schema vulnerability.



```
package owasp;
import smrl.mr.language.Action;
public class OTG_AUTHZ_002 extends MR{
  public boolean mr() {
    for (final Action action : Input(1).actions()) {
      {
        ifThenBlock();
        if (((( cannotReachThroughGUI( User(2), action.getUrl())
                && (!isSupervisorOf(User(2), action.getUser())))
                && (!isError(Output(Input(1), action.getPosition()))))
                && EQUAL(Input(2), changeCredentials(Input(1), User(2))))) {
          ifThenBlock();
          boolean _NOT = NOT( Output( Input(1), action.getPosition()).equals(
            Output( Input(2), action.getPosition())));
          if (_NOT) {
            expressionPass(); /* //PROPERTY HOLDS" */
          } else {
            return Boolean.valueOf(false);
          }
        } else {
          expressionPass(); /* //PROPERTY HOLDS" */
    }}}
    return true;
}}
```

Figure 4: Java code generated from the MR in Fig. 3.

(a) Example Crawljax Graphs



(b) Extracted Source Inputs

Data Type	Input #	Content
Input	1	A1,A2
Input	2	A1,A3
Input	3	A1,A4
Input	4	A5,A6
Input	5	A5,A7
User	1	id="admin",pwd="123"
User	2	id="devel",pwd="abc"

A1 to A7 represent different interactions, as in the following example

ID	Action Type	Element	URL	data
A1	CLICK	DIV[1]/BUTTON[1]	me.com/login	id="admin"; pwd="123"

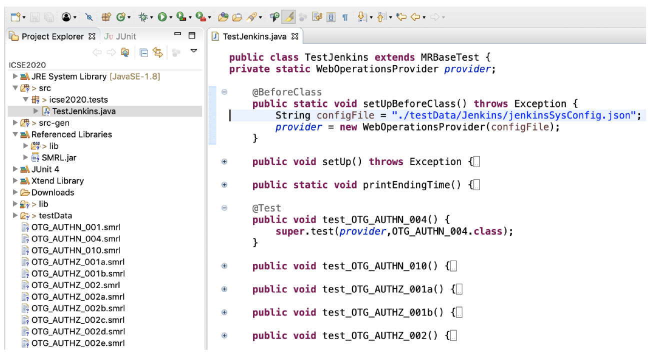
Figure 5: Example graphs and extracted data sequences.

first source input of type *Input* is a sequence of actions extracted as a path from the root to a leaf of the graphs in depth-first traversal. In our DSL, `Input(i)` returns the *i*th source input, if the same identifier has not been used to define a follow-up input.

In addition to Crawljax, our tool processes test scripts written for functional testing based on the Selenium framework [3], and derives additional source inputs. These scripts exercise complex interaction sequences not triggered by Crawljax.

3.4 Selection and Execution of MRs

To select MRs to be executed, the engineer should write a JUnit test case. The SMRL tool provides a Java class, `MRBaseTest`, which extends the JUnit framework with utility functions to facilitate the selection of MRs. Fig. 6 shows a JUnit test case selecting multiple



```
public class TestJenkins extends MRBaseTest {
private static WebOperationsProvider provider;

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
        String configFile = "./testData/Jenkins/jenkinsSysConfig.json";
        provider = new WebOperationsProvider(configFile);
    }

    public void setUp() throws Exception {

    public static void printEndingTime() {

    @Test
    public void test_OTG_AUTHN_004() {
        super.test(provider,OTG_AUTHN_004.class);
    }

    public void test_OTG_AUTHN_010() {

    public void test_OTG_AUTHZ_001a() {

    public void test_OTG_AUTHZ_001b() {

    public void test_OTG_AUTHZ_002() {
```

Figure 6: JUnit test case to select and execute MRs.

MRs (e.g., OTG_AUTHN_004 and OTG_AUTHN_010) through the function `test`. Setup methods are used to specify the configuration for the `WebOperationsProvider`, i.e., the component in charge of loading the source inputs processed by the tool.

We defined a catalog of MRs based on the security testing activities described in the OWASP book on security testing [19]. To be selected, these MRs need to be copied in the workspace (see files with extension `.smrl` in Fig. 6) and referenced in the JUnit test file.

MRs are executed as standard JUnit test classes through the Eclipse user interface or the provided console wrapper. Each MR is treated as a distinct JUnit test case. For each MR, `MRBaseTest` automatically invokes the MT algorithm [17] to execute the MR . The MT algorithm processes the bytecode of the relation to identify the types of source inputs used by the relation (e.g., Input and User). It ensures that each source input is used in at least one execution. Also, it ensures that all possible source input combinations are stressed during the execution of the relation (e.g., all available URLs with all configured users). Finally, it generates follow-up inputs from the source inputs at each invocation of the operator `EQUAL`.

If the MR does not hold, source inputs, follow-up inputs, and system outputs are shown to engineers. The tool 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.

The MT algorithm and the `MRBaseTest` class are implemented in the library `SMRL.jar`. This facilitates the integration of MT procedures into the automated build operations of software projects.

4 EVALUATION

This section provides an overview of the main findings of an evaluation conducted to address the following research questions [17]:

- *RQ1. To what extent can the SMRL tool address the oracle problem for security testing?*
- *RQ2. Is the SMRL tool effective?*

To address *RQ1*, we analyzed the security testing activities recommended by OWASP [19]. We classified the activities based on state-of-the-art oracle automation approaches (implicit oracle, catalog-based, and vulnerability-specific approaches). We identified that, out of 90 testing activities, 19 do not need an oracle, 30 are automated by state-of-the-art approaches, and 41 cannot be addressed by state-of-the-art approaches. Based on the MRs in our catalog, the SMRL tool can automate 16 (39%) of these 41 activities. The remaining 25 activities require humans to determine vulnerabilities. We conclude that the SMRL tool can thus play a key role in addressing the oracle problem in security testing.

To address *RQ2*, we selected two case studies, a commercial healthcare Web system, EDLAH2 [2], and a leading open source system, Jenkins [9]. EDLAH2 is affected by 11 vulnerabilities discovered by manual testing. Concerning Jenkins, we considered version 2.121.1. We selected all the vulnerabilities (20 in total) triggerable from the Web interface, discovered in 2018, and reported in the CVE database [1] after June 1st, 2018.

In line with the RQ1 findings, our tool addressed 36% (4 out of 11) and 40% (8 out of 20) of the vulnerabilities affecting EDLAH2 and Jenkins, respectively. We thus evaluated our tool against this 12 vulnerabilities addressed. We measured the fault detection rate (i.e., the percentage of vulnerabilities discovered) and the false positive rate (i.e., the portion of follow-up inputs leading to false alarms).

The tool achieved a very high fault detection rate when we used both Crawljax and manual test scripts for the data collection (100% for EDLAH2 and 75% for Jenkins). Overall, it detected 83.33% of the vulnerabilities targeted in our evaluation. The tool achieved a fault detection rate of 75% for EDLAH2 and of 50% for Jenkins, when we used only Crawljax for the data collection. In total, it detects 7 out of 12 vulnerabilities, 58.33%, a very promising result considering that it is almost completely automated.

The tool achieved an extremely low false positive rate (0.50%). The very small fraction of follow-up inputs leading to false alarms (32 out of 6401) show that our MRs are sound. False alarms are due to Crawljax, which, in Jenkins, did not traverse all the URLs provided by the GUI.

The SMRL tool is approximately 13k lines of code, excluding automatically generated code and third-party libraries. Additional details about the tool, including the catalog of predefined MRs, case studies, executable files and a screencast covering motivations, are available on the website at:

https://sntsvv.github.io/SMRL/

5 CONCLUSION

We presented a tool that aims to alleviate the oracle problem in security testing. The key characteristics of our tool are (1) a DSL for specifying metamorphic relations for security testing, (2) a data collection framework automatically deriving input data, and (3) a testing framework automatically performing security testing.

The tool can automate 39% of the OWASP security testing activities not currently targeted by state-of-the-art techniques, which in turn indicates that it significantly alleviates the oracle problem in security testing. Our evaluation with two case studies shows that the tool detects 83% of the targeted vulnerabilities with limited manual effort and, thus suggesting it is highly effective.

ACKNOWLEDGMENT

This work has received funding from the National Research Fund (FNR), Luxembourg, with grant INTER/AAL/15/11213850, from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277), and from the Canada Research Chair programme.

REFERENCES

[1] 2019. Common Vulnerabilities and Exposures. https://cve.mitre.org/cve/.
[2] 2019. EDLAH2. http://www.aal-europe.eu/projects/edlah2/.
[3] 2019. Selenium Web Testing Framework, https://www.seleniumhq.org/.
[4] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525.
[5] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. 2011. Finding Software Vulnerabilities by Smart Fuzzing. In *ICST'11.* 427–430.
[6] Lorenzo Bettini. 2016. *Implementing Domain-Specific Languages with Xtext and Xtend.* Packt Publishing Ltd.
[7] Wing Kwong Chan, Tsong Y Chen, Shing Chi Cheung, TH Tse, and Zhenyu Zhang. 2007. Towards the Testing of Power-aware Software Applications for Wireless Sensor Networks. In *ADA Europe'07.* 84–99.
[8] Tsong Yueh Chen, Shing-Chi Cheung, and Siu-Ming Yiu. 1998. *Metamorphic Testing: a New Approach for Generating Next Test Cases.* Technical Report. The Hong Kong University of Science and Technology.
[9] Eclipse Foundation. [n.d.]. Jenkins CI/CD server. https://jenkins.io/.
[10] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. 2012. Xbase: Implementing Domain-Specific Languages for Java. In *GPCE'12.* 112–121.
[11] Ralph Guderlei and Johannes Mayer. 2007. Towards Automatic Testing of Imaging Software by Means of Random and Metamorphic Testing. *International Journal of Software Engineering and Knowledge Engineering* 17, 6 (2007), 757–781.
[12] V. I. Levenshtein. 1966. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady* 10 (Feb. 1966).
[13] Huai Liu, Fei-Ching Kuo, Dave Towey, and Tsong Yueh Chen. 2014. How Effectively Does Metamorphic Testing Alleviate the Oracle Problem? *IEEE Transactions on Software Engineering* 40, 1 (2014), 4–22.
[14] Phu X. Mai, Arda Goknil, Lwin Khin Shar, Fabrizio Pastore, Lionel C. Briand, and Shaban Shaame. 2018. Modeling Security and Privacy Requirements: a Use Case-Driven Approach. *Information and Software Technology* 100 (2018), 165–182.
[15] Phu X. Mai, Fabrizio Pastore, Arda Goknil, and Lionel C. Briand. 2018. A Natural Language Programming Approach for Requirements-based Security Testing. In *ISSRE'18.* 58–69.
[16] Phu X. Mai, Fabrizio Pastore, Arda Goknil, and Lionel C. Briand. 2019. MCP: A Security Testing Tool Driven by Requirements. In *ICSE'19.* 55–58.
[17] Phu X. Mai, Fabrizio Pastore, Arda Goknil, and Lionel C. Briand. 2020. Metamorphic Security Testing for Web Systems. In *ICST'20.*
[18] Ali Mesbah, Arie Van Deursen, and Stefan Lenselink. 2012. Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes. *ACM Transactions on the Web* 6, 1 (2012), 3.
[19] Matteo Meucci and Andrew Muller. 2019. OWASP Testing Guide v4. https://www.owasp.org/images/1/19/OTGv4.pdf.
[20] C. Murphy, K. Shen, and G. Kaiser. 2009. Using JML Runtime Assertion Checking to Automate Metamorphic Testing in Applications without Test Oracles. In *ICST'09.* 436–445. https://doi.org/10.1109/ICST.2009.19
[21] Saahil Ognawala, Martín Ochoa, Alexander Pretschner, and Tobias Limmer. 2016. MACKE: Compositional Analysis of Low-level Vulnerabilities with Symbolic Execution. In *ASE'16.* 780–785.
[22] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortes. 2016. A Survey on Metamorphic Testing. *IEEE Transactions on Software Engineering* 42, 9 (2016), 805–824.
[23] H. Zhu. 2015. JFuzz: A Tool for Automated Java Unit Testing Based on Data Mutation and Metamorphic Testing Methods. In *TSA'15.* 8–15.