Metamorphic Security Testing for Web Systems

Phu X. Mai*, Fabrizio Pastore*, Arda Goknil*, Lionel Briand*†
*SnT Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg
†School of Engineering and Computer Science, University of Ottawa, Canada
{xuanphu.mai,fabrizio.pastore,arda.goknil,lionel.briand}@uni.lu

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

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

We provide a catalog of 22 system-agnostic MRs to automate security testing in Web systems. Our approach targets 39% of the OWASP security testing activities not automated by state-of-the-art techniques. It automatically detected 10 out of 12 vulnerabilities affecting two widely used systems, one commercial and the other open source (Jenkins).

***Index Terms*—Software Engineering, Software Security**

I. Introduction

Security testing aims to uncover flaws in software mechanisms that protect data and ensure the delivery of the intended system functionality. It is driven by security requirements which encompass both security properties of the system and the prevention of potential security threats [1]–[5]. In contexts where test case execution is automated, an automated *test oracle* (i.e., a mechanism for determining whether a test case has passed or failed) is needed to check the execution result. It often consists of comparing expected and observed outputs.

Security test cases seldom rely on automated test oracles, most often because it is infeasible or impractical to specify them due to a large number of test inputs. In other words, *security testing suffers from the oracle problem* [6]–[8], which refers to situations where it is extremely difficult or impractical to determine the correct output for a given test input. For instance, a security test case for the bypass authorization schema vulnerability should verify, for every specific user role, whether it is possible to access resources that should be available only to a user who holds a different role [9]. This type of vulnerability can often be discovered by verifying the access to various resources with different privileges and roles. However, questions arise when defining oracles. What are the resources that can only be accessible by a user with a specific role or privilege? Are the test outputs consistent with expectations regarding accessibility? In practice, it is not always feasible to answer such questions when expected outputs need to be identified for a large set of test inputs (e.g., for various resources, roles and privileges). Recent incidents involving corporate Web sites, such as Facebook's, indicate that it is particularly difficult to verify, at testing time, large sets of input sequences including the ones that trigger vulnerabilities [10], [11].

Although several security testing approaches have been proposed, they typically do not address the oracle problem and assume the availability of an implicit test oracle [6]. Furthermore, most approaches focus on a particular vulnerability (e.g., buffer overflows [12], [13]) and can only uncover vulnerabilities that prevent a system from providing results (e.g., system crashes because of buffer overflows).

Metamorphic Testing (MT) is a testing technique which has shown, in some contexts, to be very effective to alleviate the oracle problem [14], [15]. *MT is based on the idea that it may be simpler to reason about relations between outputs of multiple test executions, called metamorphic relations (MRs), than it is to specify its input-output behavior* [16]. In MT, system properties are captured as MRs that are used to automatically transform an initial set of test inputs into follow-up test inputs. If the outputs of the system under test for the initial and follow-up test inputs violate the MR, it is concluded that the system is faulty.

Considerable research has been devoted to developing MT approaches for application domains such as computer graphics (e.g., [17]–[20]), Web services (e.g., [21]–[23]), and embedded systems (e.g., [24]–[27]). Unfortunately, only a few approaches target security aspects [28]; also, their applicability is limited to the functional testing of security components (e.g., code obfuscators [28]) or to the verification of specific security bugs (e.g., heartbleed [29]). They do not support the specification of general security properties by using MRs. Although MT is automatable, very few MT approaches provide proper tool support [16]. This is also a significant obstacle for tailoring the current approaches for security testing. Our goal in this paper is to adopt MT to address the test oracle problem in security testing. Our motivation is to have a systematic way to specify MRs that capture security properties of Web systems (i.e., properties that are violated only if the system is vulnerable) and to automate security testing by relying on these MRs. An example of MR to spot bypass authorization schema vulnerabilities is: *a Web system should return different responses to two users when the first user requests a URL that is provided to her by the GUI (e.g., in HTML links) and the second user requests the same URL but this URL is not provided to her by the GUI.* In other words, a user should not

Accepted for publication in the Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST) 2020

be able to directly access URLs not provided by the GUI.

In this paper, we propose an MT approach that supports engineers in specifying MRs to capture security properties of Web systems and that automatically detects vulnerabilities (i.e., violations of security properties) based on those relations. Our approach is built on top of the following novel contributions: (1) a Domain-Specific Language (DSL) for specifying MRs for software security, (2) a catalog of system-agnostic MRs targeting well-known security vulnerabilities of Web systems [9], (3) a framework that automatically collects the data required to perform MT, and (4) a testing framework that automatically performs security testing based on the MRs and the collected data. To facilitate the specification of MRs in our DSL, we provide an editor which has been implemented as a plug-in for the Eclipse IDE [30].

We applied our approach to discover vulnerabilities in a commercial Web system and in Jenkins, a leading open source automation server [31]. The approach automatically detected 100% and 75% of the targeted vulnerabilities affecting these two systems, respectively. Based on these results and an assessment of the effort involved, we conclude that our approach is practical and beneficial to alleviate the oracle problem in security testing and to automatically detect vulnerabilities in industrial settings. Our MT toolset and the empirical data are publicly available [32].

This paper is structured as follows. Section II provides the background information regarding MT. Section III discusses the related work. In Section IV, we present an overview of the approach. Sections V to VIII describe the core technical solutions. Section IX presents our catalog of MRs. In Section X, we present the empirical evaluation of our approach. We conclude the paper in Section XI.

II. Background

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 [33].

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

As an example, let us consider an algorithm f that computes the shortest path for an undirected graph G. For any two nodes a and b in the graph G, it may not be practically feasible to generate all possible paths from a to b, and then check whether the output path is really the shortest path. However, a property of the shortest path algorithm is that the length of the shortest path will remain unchanged if the nodes a and b are swapped. Using this property, we can derive an MR, i.e., $|f(G, a, b)| = |f(G, b, a)|$, in which we need two executions of the function under test, one with (G, a, b) and another one with (G, b, a). The results of the two executions are verified against the relation. If there is a violation of the relation, then f is faulty.

We provide below basic definitions underpinning MT.

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

In our example, the property of the target algorithm f is "the length of the shortest path will remain the same if the start and end nodes are swapped". The MR of this property is $(x_1 = (G, a, b)) \wedge (x_2 = (G, b, a)) \rightarrow |f(x_1)| = |f(x_2)|$.

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

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

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

where $swapLastArguments$ is the transformation function.

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

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

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

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

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

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

III. Related Work

Security testing approaches can be categorized [3] as follows: (1) security functional testing validating whether the specified security properties are implemented correctly, and (2) security vulnerability testing simulating attacks that target typical system vulnerabilities.

Many vulnerability testing approaches rely on an implicit test oracle, i.e., one that relies on implicit knowledge to distinguish between correct and incorrect system behavior [1]. This is the case for approaches targeting buffer overflows, memory leaks, unhandled exceptions, and denial of service [13], [34], [35], most of which rely on mutational fuzzing [36], i.e., the generation of new inputs through the random modification of existing inputs. Implicit oracles deal with simple abnormal system behavior such as unexpected system termination and are not system-agnostic. What is abnormal in one system might be considered normal in a different context [6].

Vulnerability testing approaches for code injections also suffer from the oracle problem [37]–[44]. To resolve this problem, Huang et al. [45] proposed an MT-like technique which sends multiple HTTP requests, i.e., one request with an injection, an intentionally invalid request, and a valid request. They compare the responses to determine if the request with the injection is filtered. Unfortunately, MT-like approaches that address a broader set of security vulnerabilities are missing.

Model-based approaches [46], [47] typically target security vulnerability testing (e.g., [48]–[64]) whereas a few solutions address security functional testing (e.g., [65]–[68]). Most of these approaches only generate test sequences from security models and do not address the oracle problem. Approaches that generate test cases including oracles [63], [64], [68] rely on mappings between model-level abstractions (i.e., tokens in markings of PrT networks) and executable code implementing the oracle logic (e.g., searching for error messages in system output). Unfortunately, these approaches do not free engineers from implementation effort since they require the manual implementation of the executable oracle code. Furthermore, the model-based mapping supported by these approaches does not enable engineers to specify precise test oracles (e.g., oracles that verify the exact content of the output of the system with respect to its inputs [63]).

With MT, we aim to address the limitations of security testing approaches. Indeed, MT supports oracle automation thanks to MRs that can precisely capture the relations between inputs and outputs. Considerable research has been devoted to developing MT approaches for various domains such as computer graphics (e.g., [17]–[20]), simulation (e.g., [69]–[71]), Web services (e.g., [21]–[23]), embedded systems (e.g., [24]–[27]), compilers (e.g., [72], [73]), and machine learning (e.g., [74], [75]). Preliminary applications of MT to security testing [28] focus on the functional testing of security components (i.e., verifying the output of code obfuscators and the rendering of login interfaces) and the verification of low level properties broken by specific security bugs (e.g., heartbleed [29]). Although these works show the feasibility of MT for security, they focus on a narrow set of vulnerabilities and do not automate the generation of executable metamorphic test cases, which are manually implemented based on the identified MRs.

Although MT is highly automatable, very few approaches provide proper tool support enabling engineers to write system-level MRs [16]. They require that MRs be defined either as Java methods [76] or pre-/post-conditions [77], which limit the adoption of MT to verify system-level, security properties. Furthermore, since MRs are often specified by capturing properties using a declarative notation, the use of an imperative language to implement the relations may force engineers to invest additional effort to translate abstract, declarative MRs.

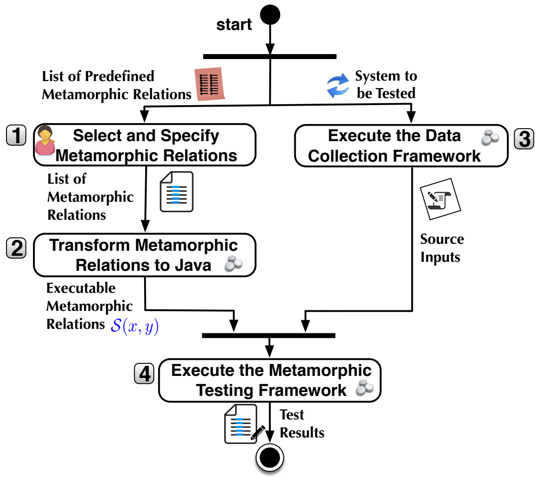

start
List of Predefined Metamorphic Relations
System to be Tested
1
Select and Specify Metamorphic Relations
Execute the Data Collection Framework
3
List of Metamorphic Relations
2
Transform Metamorphic Relations to Java
Source Inputs
Executable Metamorphic Relations $\mathcal{S}(x,y)$
4
Execute the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

Fig. 1. Overview of the approach.

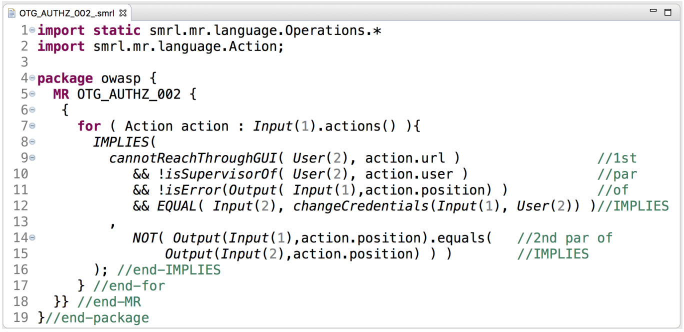
To summarize, existing automated security testing approaches lack support for the generation of test oracles. The few approaches addressing the oracle problem either focus on a limited set of security vulnerabilities, or integrate oracles with limited capabilities. MT can overcome these limitations.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

IV. Overview of the Approach

The process in Fig. 1 presents an overview of our approach. In Step 1, the engineer selects, from a catalog of predefined MRs, the relations for the system under test. We have derived our catalog of MRs from the testing guidelines [9] edited by OWASP [78]. In addition, the engineer can also specify new relations by using our DSL. Step 1 is manual. We discuss this step in Section V. In Step 2, our approach automatically transforms the MRs into executable Java code (Section VI).

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

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



```
 1 import static smrl.mr.language.Operations.*
 2 import smrl.mr.language.Action;
 3
 4 package owasp {
 5   MR OTG_AUTHZ_002 {
 6    {
 7      for ( Action action : Input(1).actions() ){
 8        IMPLIES(
 9          cannotReachThroughGUI( User(2), action.url )                //1st
10             && !isSupervisorOf( User(2), action.user )               //par
11             && !isError(Output( Input(1),action.position) )          //of
12             && EQUAL( Input(2), changeCredentials(Input(1), User(2)) )//IMPLIES
13          ,
14             NOT( Output(Input(1),action.position).equals(   //2nd par of
15                 Output(Input(2),action.position) ) )        //IMPLIES
16        ); //end-IMPLIES
17      } //end-for
18   }} //end-MR
19 }//end-package
```

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

TABLE I
EXCERPT OF THE DATA FUNCTIONS IN SMRL.

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

Our DSL and the data collection framework can be extended to support new language constructs and data collection methods. The MT framework can be extended to deal with input interfaces not supported yet (e.g., Silverlight plug-ins [79]) and to load data collected by new data collection methods.

V. SMRL: A DSL FOR METAMORPHIC RELATIONS

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

SMRL is an extension of Xbase [80], an expression language provided by Xtext [81]. Xbase specifications can be translated to Java programs and compiled into executable Java bytecode. We rely on Xbase since DSLs extending Xbase inherit the syntax of a Java-like expression language as well as language infrastructure components, including a parser, a linker, a compiler and an interpreter [80]. These features will facilitate the adoption of SMRL.

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

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

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

A. SMRL Grammar

The SMRL grammar extends the Xbase grammar, which extends the Java grammar. Each SMRL specification can have an arbitrary number of import declarations which indicate the APIs to be used in MRs (Line 1 in Fig. 2).

A package declaration resembles the Java package structure and can contain one or more MRs. Line 4 in Fig. 2 declares the package *owasp*, which is is the package for our MRs. Like in Java, MRs defined in different SMRL specification files can belong to the same package.

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

B. Data Representation Functions

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

Each data function returns a data class instance. Fig. 3 presents the SMRL data model where all classes are subtypes of either `InputType` or `OutputType`. `InputType` represents input data that can be defined to trigger a certain system behavior. `InputSequence` represents a sequence of interactions between a user and the system under test and is consequently associated with `Action`. `Action` represents an activity performed by a user (e.g., requesting a URL). It carries information about actions such as a URL requested by an action and parameters in the URL query string. `Action` is associated with `Session`, which represents a user session in a Web application. `User` represents a system user.

A *source input* is an instance of `InputType` returned by one of the data functions; a *follow-up* input is an instance of `InputType` modified by means of a Web-specific function (see Section V-D). For example, a source input might be a sequence of two HTTP requests for user login and user profile visualization. A follow-up input is the same sequence with login credentials for a different user. Instances of `OutputType` capture outputs generated by the system when processing an input; each instance of `OutputType` is associated with an instance of `InputType`. The last three functions in Table I return predefined/random values. They are used to redefine attributes of follow-up inputs as described in Section VIII.

C. Boolean Operators

SMRL provides seven boolean operators, i.e., `IMPLIES`, `AND`, `OR`, `TRUE`, `FALSE`, `NOT` and `EQUAL`. They enable the

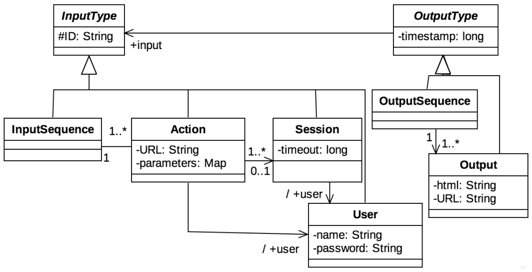


Fig. 3. Metamorphic data classes in SMRL.

definition of *metamorphic expressions*, which are boolean expressions that should hold for an MR to be true. A *metamorphic expressions* is a specific kind of `XBlockExpression`. We use metamorphic expressions to decompose an MR into simple properties. They are defined in a declarative manner, which is standard practice in MT.

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

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

D. Web-Specific Functions

MRs for security testing often capture complex properties of Web systems that cannot be expressed with simple boolean or arithmetic operators. Therefore, SMRL provides a set of functions that capture typical properties of Web systems and alter Web data. Table II describes a portion of the 30 Web-specific functions in SMRL [32]. Each function is provided as a method of the SMRL API. Engineers can specify additional functions as Java methods. The new functions can be used in SMRL thanks to the underlying Xtext framework.

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

TABLE II
EXCERPT OF THE WEB-SPECIFIC FUNCTIONS IN SMRL.

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

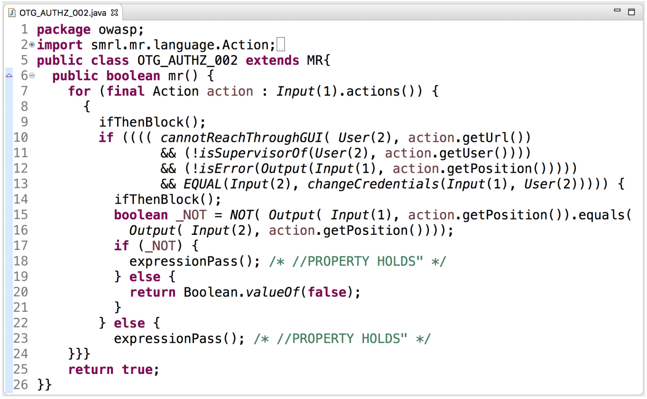
In Fig. 2, the function `cannotReachThroughGUI` checks if the URL of the current action cannot be reached from the GUI (Line 9). The function `isSupervisorOf` checks if `User(2)` is not a supervisor of `User(1)` (Line 10). The function `isError` returns true if an output page contains an error message,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

VI. SMRL TO JAVA TRANSFORMATION

SMRL specifications are automatically transformed into Java code (Step 2 in Fig. 1). To this end, we extended the Xbase compiler (hereafter SMRL compiler). Each MR is transformed into a Java class with the name of the relation and its package. The generated classes extend the class `MR` and implement its method `mr`.

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

Fig. 4 shows the Java code generated from the relation in Fig. 2. A loop control structure is generated from the loop instruction in the relation (Line 7). The loop body contains the Java code generated from the metamorphic expression using the operator `IMPLIES` (Lines 10-24). The first `IF` condition



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

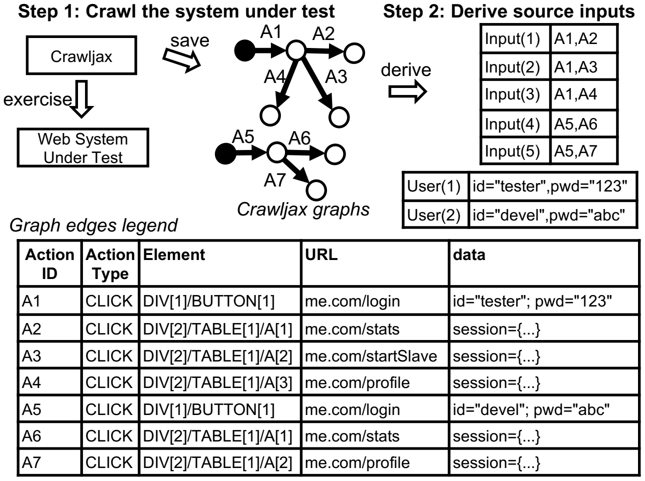
Fig. 4. Java code generated from the MR in Fig. 2.

checks whether the first parameter of the operator IMPLIES holds (Lines 10-13). The nested IF block checks whether the second parameter of IMPLIES holds (Line 17). If the expression does not hold, mr returns false (Line 20). The relation holds only if all the expressions in the loop hold. Therefore, the SMRL compiler generates a return true statement after the loop body (Line 25). Calls to the methods ifThenBlock and expressionPass are used to erase the generated follow-up inputs at each iteration.

VII. Data Collection Framework

To automatically derive source inputs (Step 3 in Fig. 1), we extended the Crawljax Web crawler [82], [83]. Crawljax explores the user interface of a Web system (e.g., by requesting URLs in HTML anchors or by entering text in HTML forms). It generates a graph whose nodes represent the system states reached through the user interface and edges capture the action performed to reach a given state (e.g., clicking on a button). Crawljax detects states based on the content of the displayed page. Our extension relies on the edit distance to distinguish system states [84]. We keep a cache of the HTML page associated to each state detected by Crawljax. When a new page is loaded, our extension computes the edit distance between the loaded page and all the pages associated to the different system states. When the distance is below a given threshold (5% of the page length), we assume that two pages belong to the same state. If a page does not belong to any state, Crawljax adds a new state to the graph. Crawling stops when no more states are encountered or a timeout is reached.

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



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.

Fig. 5 exemplifies the data collection steps. First, Crawljax generates the graphs of the system under test. Second, source inputs are automatically derived from the graphs. For example, an input sequence is a path from the root to a leaf of a Crawljax graph in depth-first traversal. The source inputs are later queried by the SMRL functions (see Section VIII). For example, Input(i) returns the i^{th} input sequence; User(i) returns the i^{th} unique login credentials in the input sequences.

In addition to Crawljax, our toolset also processes manually implemented test scripts to generate additional source inputs. It processes test scripts based on the Selenium framework [85] and derives a source input from each. We rely on test scripts to exercise complex interaction sequences not triggered by Crawljax (see Section X). Crawljax, instead, performs an almost exhaustive exploration of the Web interface, which is typically not done by test scripts. Engineers can reuse scripts developed for functional testing, or define new ones.

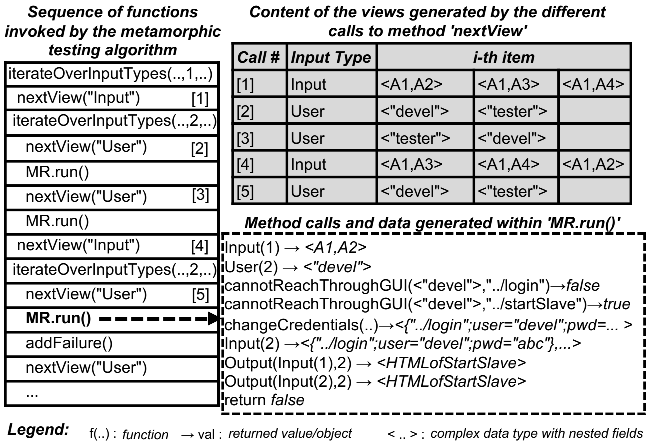
VIII. Metamorphic Testing Framework

We automatically perform testing based on the executable MRs in Java and the data collected by the data collection framework (Step 4 in Fig. 1). Fig. 6 presents our testing algorithm. The algorithm takes as input a MR and a data provider exposing the collected data (source inputs). We first process the bytecode of the MR to identify the types of source inputs referenced by the relation (e.g., *Input* and *User*). This is achieved by the function extractSourceInputTypes (Line 2) which identifies the calls to the *data representation functions* using the ASM static analysis framework [86]. We ensure that all possible combinations of available source inputs are stressed during the execution of the relation (e.g., we would like to access all available URLs with all configured users). This is achieved by the function iterateOverInputTypes (Line 3). The function iterates over all available items for a given input type (e.g., all available users) and is recursively invoked for each input type in the MR.

The function iterateOverInputTypes is driven by the methods exposed by the data provider (Lines 7 and 8). The

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

Fig. 6. Metamorphic testing algorithm.


Sequence of functions invoked by the metamorphic testing algorithm
iterateOverInputTypes(..,1,..)
nextView("Input") [1]
iterateOverInputTypes(..,2,..)
nextView("User") [2]
MR.run()
nextView("User") [3]
MR.run()
nextView("Input") [4]
iterateOverInputTypes(..,2,..)
nextView("User") [5]
MR.run()
addFailure()
nextView("User")
...
Content of the views generated by the different calls to method 'nextView'

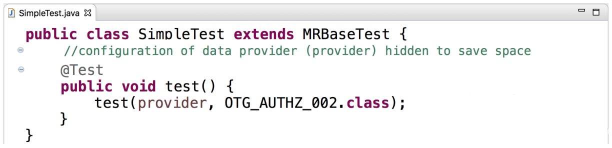
Call #	Input Type	i-th item		
[1]	Input	<A1,A2>	<A1,A3>	<A1,A4>
[2]	User	<"devel">	<"tester">	
[3]	User	<"tester">	<"devel">	
[4]	Input	<A1,A3>	<A1,A4>	<A1,A2>
[5]	User	<"devel">	<"tester">	

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
```

Legend: f(..) : function → val : returned value/object < .. > : complex data type with nested fields

Fig. 7. Data processing for the relation in Fig. 2.



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

Fig. 8. Example metamorphic test case. Engineers need only to configure the data provider and select the MR (s) to be tested.

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

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

When the relation does not hold (Lines 13 and 14), the function `addFailure` stores the failure context information (i.e., source-inputs, follow-up inputs, and system outputs). To minimize the time spent by engineers in analyzing failures triggered by distinct follow-up inputs exercising a same vulnerability, we report only failures that perform HTTP requests (e.g., accessing a URL) not generated by input sequences that led to previously reported failures.

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

Fig. 7 exemplifies 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. 7) because the *login* and *stats* pages have been accessed by both users *devel* and *tester* and thus the implication holds. The third invocation of `MR.run` returns false because the output page for the *startSlave* URL is the same for the two input sequences and thus the relation does not hold. To determine if Web pages are equal, we rely on edit distance. Our framework relies on JUnit [87] to integrate MT into traditional testing environments (see Fig. 8).

IX. Catalog of Metamorphic Relations

We derived a catalog of MRs from the activities described in the OWASP book on security testing [9]. The book provides detailed descriptions of 90 testing activities (hereafter *OWASP testing activities*) for Web systems; each OWASP testing activity targets a specific vulnerability. For example, for the bypass authorization schema vulnerability, OWASP suggests to collect links in administrative interfaces and to directly access the corresponding URLs by using credentials of other users. Based on this suggestion, we defined the MR in Fig. 2.

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

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

Table III presents an excerpt of our catalog along with a description of each MR. The full catalog of MRs is available for download [32]. All the MRs in the catalog are expressed by

TABLE III
EXCERPT OF THE METAMORPHIC RELATION CATALOG FOR SECURITY TESTING.

OTG-AUTHN-001: Testing for credentials transported over an encrypted channel
MR OTG_AUTHN_001 {
 {
    for (Action action : Input(1).actions()) {
        var pos = action.getPosition();
        IMPLIES(
            isLogin(action)   //1st par (1st clause)
              && EQUAL (Input(2) , Input(1))   //1st par (2nd clause)
              && Input(2).actions.get(action.position).setChannel("http") //1st par (3rd clause)
            ,
            different (Output(Input(1),pos), Output(Input(2),pos))   //2nd par of IMPLIES
        );//end-IMPLIES
    }//end-for
 }
}//end-MR
	Description: A login operation should not succeed if performed on the http channel. The 1st parameter of the operator `IMPLIES` is a boolean expression with three clauses joined with logical conjunctions. The 1st clause checks if the current action performs a login. The 2nd clause defines the follow-up input. The 3rd clause changes the channel of the login action in the follow-up input. The 2nd parameter of IMPLIES checks if the output generated by the login operation is different in the two cases.

OTG-AUTHZ-001: Testing for directory traversal/file include
 	
MR OTG_AUTHZ_001 {
 {
    for (Action action : Input(1).actions()){
        for (var par=0; par < action.getParameters().size(); par++){
            var pos = action.getPosition();
            IMPLIES(
                EQUAL(Input(2), Input(1)) //1st par of IMPLIES (1st clause)
                  && Input(2).actions().get(pos).setParameterValue(par, RandomFilePath())//(2nd clause)
                ,
                OR(//2nd par of IMPLIES, OR operator receiving 2 parameters
                  isError(Output(Input(2),pos))   //1st par of OR
                  ,
                  userCanRetrieveContent(action.getUser(), Output(Input(2),pos)))//2nd par of OR
            );//end-IMPLIES
        }//end-for
    }//end-for
 }
}//end-MR
}
	Description: A file path passed in a parameter should never enable a user to access data that is not provided by the user interface. This metamorphic relation contains two nested loops; the first iterates over the actions in the input sequence, the second iterates over the parameters of the action. The 1st parameter of the operator `IMPLIES` is a boolean expression with two clauses joined with a logical conjunction. The 1st clause defines a follow-up input that is a copy of the source input. The 2nd clause set the value of a parameter to a random file path. The 2nd parameter of IMPLIES verifies the result. It is implemented as an OR operation where the 1st parameter verifies that the follow-up input leads to an error page. The 2nd parameter deals with the case in which the generated request is valid, and verifies that the returned content is something that the user has the right to access. The framework evaluates the MR as many times as needed to provide 100 different random file paths to the parameters of the action in the position `pos`.

OTG-SESS-003: Testing for session fixation
MR OTG_SESS_003 {
 {
    for(Action signup : Input(1).actions()){
        for (Action f : Input(2).actions()) {
            var pos = f.getPosition();
            IMPLIES(
                isSignup(signup) && //1st par of IMPLIES (1st clause)
                afterLogin(f) && //1st par of IMPLIES (2nd clause)
                EQUAL(Input(3), addAction(Input(2), pos+1, signup)) //(3rd clause)
                ,
                different(Output(Input(3), pos).getSession(), //2nd par of IMPLIES
                           Output(Input(3), pos+1).getSession())
            );//end-IMPLIES
        }//end-for
    }//end-for
 }
}//end-MR
	Description: A signup action should always lead to a new session ID, even when performed by a user who is already logged-in. This metamorphic relation contains two nested loops iterating over the actions of two distinct source input sequences (i.e., `Input(1)` and `Input(2)`). The first loop looks for a signup action (i.e., 'signup'), the second looks for an action (i.e., 'f') following a login. The 1st parameter of the operator `IMPLIES` is a boolean expression with three clauses joined with a logical conjunction. The 1st clause checks if we are in the presence of a signup action. The 2nd clause checks if the action 'f' follows a login. The 3rd clause defines a follow-up input by copying the signup action after the action 'f' in the source input `Input(2)`. The 2nd parameter of IMPLIES verifies the result by checking that the session ID following the signup action is different than the one of the previous page.

Notes: Our catalog of metamorphic relations covers also the following OWASP activities: testing for HTTP Strict Transport Security (OTG-CONFIG-007), testing for weaker authentication in alternative channel (OTG-AUTHN-010), testing for privilege escalation (OTG-AUTHZ-003), testing for bypassing authentication schema (OTG-AUTHN-004), testing for insecure direct object references (OTG-AUTHZ-004), testing for logout functionality (OTG-SESS-006), test session timeout (OTG-SESS-007), testing for Session puzzling (OTG-SESS-008), testing for HTTP verb tampering (OTG-INPVAL-003), testing for HTTP parameter pollution (OTG-INPVAL-004), testing for weak encryption (OTG-CRYPST-004), test number of times a function can be used (OTG-BUSLOGIC-005), test for bypass authorization schema (OTG-AUTHZ-002, see Fig. 2).

means of an implication (the operator `IMPLIES`). The operator `EQUAL` is used to define follow-up inputs. It indicates that the follow-up input (typically `Input(2)`) is a copy of the source input (usually `Input(1)`) except for the differences made by the function calls following the operator. For example, in `OTG_AUTHN_001`, the follow-up input is equal to the source input except for one action of the input sequence which should be performed on the HTTP channel.

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

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

X. EVALUATION

Our evaluation addresses the following research questions:

RQ1. To what extent can metamorphic testing address the oracle problem in the context of security testing? We aim to determine which types of security vulnerabilities can be addressed by our solution.

RQ2. Is the proposed solution effective? The goal is to assess whether the proposed solution enables, in a reliable manner, the automated detection of security vulnerabilities.

RQ1 To answer RQ1, we analyzed the security testing activities recommended by OWASP [9]. For each activity, we identified state-of-the-art oracle automation strategies. Table IV lists the number of activities automated by these strategies.

Implicit oracle. Some activities can be automated by random test input generation strategies relying on implicit oracles. For instance, *testing for buffer overflow* [88] is automated by looking for system crashes in response to lengthy inputs.

Catalog-based. We can automate some activities based on a predefined catalog in which we specify inputs and oracles. For instance, we can use a catalog to perform a dictionary attack for *testing for default credentials* [89].

TABLE IV
ORACLE AUTOMATION STRATEGIES FOR SECURITY TESTING*

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

*Details are available online [32].

TABLE V
VULNERABILITY TYPES ADDRESSED BY SMRL MRS *

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

No oracle needed. Some activities collect data to reverse engineer the system under test. They do not verify security properties of the system and thus do not have an oracle problem. For instance, the activity *mapping application architecture* [90] identifies the components of a Web system.

Manual oracle. Some activities require humans to determine vulnerabilities based on system specifications. For instance, when *testing for the circumvention of work flows* [91] on pay-per-view systems, only a human can decide if pending transactions should grant service access, based on specifications.

Vulnerability-specific approaches. Some activities can be automated by state-of-the-art tools such as Burp Suite (BS) [92] and thus may not necessarily benefit from MT. These are the OWASP testing activities that detect cross site scripting and code injection vulnerabilities. Other activities are either not targeted or partially automated. For example, BS does not automate oracles for OTG-AUTHZ-002 [93]. BS enables engineers to compare the content of site maps [94] recorded in different user sessions (e.g., with and without certain privileges). Unfortunately, it requires that engineers manually identify the privileged resources and inspect the differences in the observed system outputs, which is error prone (e.g., overlooking privileged resources) and expensive. Even BS plug-ins using Crawljax to build site maps do not address the oracle problem but generate JUnit tests that simply retrieve the mapped resources [95]. With SMRL, engineers, instead, can focus on the specification of system-level properties without performing manual testing activities. Testing activities, including oracles, are automated by the MT framework.

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

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

To further characterize our catalog of MRs, we report in Table V the number of MRs targeting the vulnerability types in the OWASP top ten list [96]. The MRs in our catalog can discover five of these ten vulnerability types, and thus *have a broad applicability scope*. Note that MRs can discover injection vulnerabilities [45] and, potentially, also XSS and XEE because they all concern injected code. In this paper we specifically target vulnerabilities not addressed by existing oracle automation approaches, which is the reason why we ignored injections. We leave the investigation of other vulnerability types to future work.

RQ2 We applied the proposed approach to discover vulnerabilities in two case studies: a commercial Web system developed in the context of the EDLAH2 project [97] (hereafter, E2) and Jenkins [31], an open source system. E2 is the entry point of a healthcare service developed by our industry partner [98]. It relies on mobile and wearable devices to support elderly people (patients) in their daily life. The E2 Web interface enables carers (e.g., family and doctors) to monitor patients' conditions. The second case study, Jenkins, is an open-source continuous integration server. We chose Jenkins since it is widely adopted and well tested. Its Web interface includes advanced features such as Javascript-based login and AJAX interfaces. The two case studies are therefore very different and provide complementary perspectives. E2 is developed in PHP [99] and based on the Drupal content management system [100]; Jenkins is a Java Web application that can be executed within any servlet container [101]. We used the latest E2 version and Jenkins version 2.121.1. We selected the Jenkins version affected by all the vulnerabilities triggerable from the Web interface, discovered in 2018, and reported in the CVE vulnerability database [102] after June 1st, 2018. Jenkins 2.121.1 is affected by 20 such vulnerabilities. E2 is affected by 12 vulnerabilities discovered by manual testing following the OWASP guidelines [5].

Our approach addresses 36% (4 out of 11) and 40% (8 out of 20) of the vulnerabilities affecting E2 and Jenkins, respectively. This is consistent with our analysis in RQ1.

For each system under test, we configured our data collection framework with multiple users having different roles. We used two credentials for E2 and four credentials for Jenkins. For each role, we executed the data collection framework to crawl the system under test for a maximum of 300 minutes. In total, the data collection took 1000 minutes for Jenkins and 40 minutes for E2. For E2 and for the anonymous role in Jenkins, Crawljax completed in less than 300 minutes because all states were visited. 73 and 156 input sequences were identified for E2 and Jenkins, respectively. Also, we implemented Selenium-based test scripts to exercise use cases not covered by Crawljax. This led to one and two test scripts for E2 and Jenkins, respectively. We tested the two systems against the MRs that target the vulnerabilities affecting them (4 for E2 and 8 for Jenkins). Our replicability package [32]

TABLE VI
SUMMARY OF RQ2 RESULTS GROUPED BY DATA COLLECTION METHOD.

Case study	Vulnerabilities	Crawljax		Crawljax & Manual	
		Specificity	Sensitivity	Specificity	Sensitivity
E2	4 [5]	100.00%	75.00%	100.00%	100.00%
Jenkins 2.121.1	8 [103]–[110]	99.34%	50.00%	99.43%	75.00%
Overall	12	99.43%	58.33%	99.50%	83.33%

does not include E2 data because of confidentiality restrictions. Comparing with state-of-the-art tools is infeasible because they do not provide automated oracles.

We measured specificity and sensitivity [111]. Specificity (i.e., the true negative rate) is the ratio of follow-up inputs, generated by our framework, that do not trigger any vulnerability and (correctly) do not lead to any MT failure. In other words, *1 - specificity* measures the time spent by engineers on unwarranted MT failures. Sensitivity (i.e., the true positive rate) is the ratio of vulnerabilities being discovered. Based on the existing vulnerability reports for the two systems considered, we identified the inputs that should uncover vulnerabilities. MT failures are expected for these inputs to be true positives. For each MT failure, we manually verified if the test input actually triggered any vulnerability (true positive). Table VI summarizes the results obtained with different data collection methods (i.e., based on Crawljax only or integrating Crawljax and manual test scripts). Each MR was tested in less than 12 hours, except one which was stopped after 24 hours. Performance optimizations are part of our future work.

We observe that the approach has **extremely high specificity** (99.50%), which indicates that only a negligible fraction of follow-up inputs inspected lead to false alarms (32 out of 6401, ∼0.5%). False alarms are due to limitations in Crawljax, which, in the case of Jenkins, did not traverse all the URLs provided by the GUI, for all the users. Consequently, MRs concerning authorization vulnerabilities fail. However, it is easy to determine that the URLs causing the false alarms should be accessible to all the users.

Sensitivity is high when data collection is based on both Crawljax and manual test scripts (100% for E2 and 75% for Jenkins). Since sensitivity reflects the fault detection rate (i.e., the portion of vulnerabilities discovered), we conclude that our approach is **highly effective**. Overall, it detects 83.33% of the vulnerabilities targeted in our evaluation. More precisely, the approach identifies 47 distinct inputs sequences triggering these vulnerabilities. The approach misses two of the eight targeted vulnerabilities in Jenkins. One of them can be detected only if the server configuration is modified during test execution [105], which is not supported by our toolset. The other one cannot be reproduced since it concerns the termination of Jenkins' reboot [103], which is not interruptible when Jenkins is not overloaded (our case).

When the data collection relies on Crawljax only, sensitivity drops below 75% for Jenkins. This occurs since Jenkins requires quick system interactions to exercise certain features (e.g., first writing a valid Unix command in a textbox to enqueue a batch job, and then quickly pressing a button to delete it from the queue). However, even when the data collection is based on Crawljax only, the overall fault detection rate is satisfactory (i.e., 58.33%), with 7 out of 12 vulnerabilities being detected. Automatically detecting 58.33% of the vulnerabilities not targeted by state-of-the-art approaches, without the need for any manual test script, is encouraging.

The benefits of our approach mostly stems from the MRs in our catalog being reusable to test any Web system. Furthermore, the required manual test scripts are few and inexpensive to implement. For the Web systems above, we manually wrote three test scripts which only include 10 actions in total. This is very limited in comparison to the total of 6401 inputs sequences (41834 actions) automatically generated by our approach to test the two systems. A traditional way to verify the same scenarios would require 6401 manually implemented test scripts, each providing a distinct input sequence, and a dedicated oracle (e.g., an assertion statement). Therefore, we conclude that our approach provides an advantageous cost-effectiveness trade-off compared to current practice.

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

XI. CONCLUSION

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

Our contributions include (1) a DSL and supporting tools for specifying MRs for security testing, (2) a set of MRs inspired by OWASP guidelines, (3) a data collection framework crawling the system under test to automatically derive input data, and (4) a testing framework automatically performing security testing based on the MRs and the input data [32].

Our analysis of the OWASP guidelines shows that our approach can automate 39% of the security testing activities not currently targeted by state-of-the-art techniques, which indicates that the approach significantly contributes to addressing the oracle problem in security testing. Our empirical results with two commercial and open source case studies show that the approach requires limited manual effort and detects 83% of the targeted vulnerabilities, 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] 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] H. Mouratidis and P. Giorgini, "Secure tropos: a security-oriented extension of the tropos methodology," *International Journal of Software Engineering and Knowledge Engineering*, vol. 17, no. 2, pp. 285–309, 2007.

[3] 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.

[4] 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.

[5] 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.

[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://www.theverge.com/2018/11/13/18088904/imperva-facebook-data-vulnerability-user-friends-information-cambridge-analytica.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

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

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

[32] Authors of this paper, "SMRL editor executable, catalog of MRs, MT framework, experimental data." https://sntsvv.github.io/SMRL/.

[33] 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.

[34] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "Finding software vulnerabilities by smart fuzzing," in *ICST'11*, 2011, pp. 427–430.

[35] A. Takanen, J. D. Demott, C. Miller, and A. Kettunen, *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2018.

[36] 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/

[37] S. Raghavan and H. Garcia-Molina, "Crawling the hidden web," in *VLDB'01*, 2000, pp. 129–138.

[38] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic, "SecuBat: A web vulnerability scanner," in *WWW'06*, 2006, pp. 247–246.

[39] 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.

[40] 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.

[41] 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.

[42] M. Salas and E. Martins, "Security testing methodology for vulnerabilities detection of XSS in web services and WS-security," *ENTCS*, pp. 133–154, 2014.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] M. Felderer, B. Agreiter, P. Zech, and R. Breu, "A classification for model-based security testing," in *VALID'11*, 2011, pp. 109–114.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] J. Jürjens, "UMLsec: Extending UML for secure systems development," in *UML'02*, 2002, pp. 412–425.

[53] ——, "Sound methods and effective tools for model-based security engineering with UML," in *ICSE'05*, 2005, pp. 322–331.

[54] ——, *Secure Systems Development with UML.* Springer Science & Business Media, 2005.

[55] ——, "Model-based security testing using UMLsec: A case study," *Electronic Notes in Theoretical Computer Science*, vol. 220, no. 1, pp. 93–104, 2008.

[56] 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.

[57] 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.

[58] E. Martin and T. Xie, "A fault model and mutation testing of access control policies," in *WWW'07*, 2007, pp. 667–676.

[59] ——, "Automated test generation for access control policies via change-impact analysis," in *SESS'07*, 2007.

[60] E. Martin, T. Xie, and T. Yu, "Defining and measuring policy coverage in testing access control policies," in *ICICS'06*, 2006, pp. 139–158.

[61] G. Wimmel and J. Jürjens, "Specification-based test generation for security-critical systems using mutations," in *ICFEM'02*, 2002, pp. 471–482.

[62] J. Whittle, D. Wijesekera, and M. Hartong, "Executable misuse cases for modeling security concerns," in *ICSE'08*, 2008, pp. 121–130.

[63] 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.

[64] 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.

[65] Y. Le Traon, T. Mouelhi, and B. Baudry, "Testing security policies: Going beyond functional testing," in *ISSRE'07*, 2007, pp. 93–102.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 216–226, 2014.

[74] 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.

[75] C. Murphy, G. E. Kaiser, and L. Hu, "Properties of machine learning applications for use in metamorphic testing," Columbia University, Tech. Rep., 2008.

[76] 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.

[77] 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.

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

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

[80] 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.

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

[82] 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.

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

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

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

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

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

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

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

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

[91] ——, "OTG-BUSLOGIC-006: Testing for the circumvention of work-flows." https://www.owasp.org/index.php/Testing_for_the_Circumvention_of_Work_Flows_(OTG-BUSLOGIC-006).

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

[93] ——, "Using burp suite to test for by-pass 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.

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

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

[96] "OWASP Top 10 Web Security Risks with weak-nesses descriptions matching the mitre cwe categories." https://cwe.mitre.org/data/definitions/1026.html.

[97] "EDLAH2: Active and Assisted Living Programme," http://www.edlah2.eu/.

[98] "MiCare," https://getmicare.co.uk/.

[99] The PHP Group, "Php programming language." http://php.net/.

[100] Drupal, "Drupal content management system." https://www.drupal.org/.

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

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

[103] MITRE, "CVE-2018-1999047, concerns OTG-AUTHZ-002," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999047.

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

[105] ——, "CVE-2018-1999045, concerns OTG-AUTHZ-002," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999045.

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

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

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

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

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

[111] D. Lane, "Online statistics education: A multimedia course of study." [Online]. Available: http://onlinestatbook.com/