A Search-based Testing Approach for XML Injection Vulnerabilities in Web Applications

Sadeeq Jan*, Cu D. Nguyen*†, Andrea Arcuri*‡, Lionel Briand*
* Interdisciplinary Centre for Security, Reliability and Trust (SNT), University of Luxembourg
† Cyber Security Department, POST Luxembourg, ‡ Westerdals ACT, Norway
{sadeeq.jan, lionel.briand}@uni.lu, duycu.nguyen@post.lu, arcand@westerdals.no

***Abstract*—In most cases, web applications communicate with web services (SOAP and RESTful). The former act as a front-end to the latter, which contain the business logic. A hacker might not have direct access to those web services (e.g., they are not on public networks), but can still provide malicious inputs to the web application, thus potentially compromising related services. Typical examples are XML injection attacks that target SOAP communications. In this paper, we present a novel, search-based approach used to generate test data for a web application in an attempt to deliver malicious XML messages to web services. Our goal is thus to detect XML injection vulnerabilities in web applications. The proposed approach is evaluated on two studies, including an industrial web application with millions of users. Results show that we are able to effectively generate test data (e.g., input values in an HTML form) that detect such vulnerabilities.**

I. Introduction

The Service-Oriented Architecture (SOA) enables modular design [1]. It brings a great deal of flexibility to modern systems and allows them to orchestrate services from different vendors. A typical SOA system consists of front-end web applications, intermediate web services, and back-end databases; they work in a harmonized fashion from the front-ends receiving user inputs, the services exchanging and processing messages, to the back-ends storing data. The Extensible Markup Language (XML) and its corresponding technologies, such as XML Schema Validation and XPath/XQuery [2], are important in SOA. Unfortunately, XML comes with a number of known vulnerabilities, such as *XML Billion Laughs (BIL)* and *XML External Entities (XXE)* [3], [4], which malicious attackers can exploit, thus compromising SOA systems. It is therefore crucial to carry out security testing to detect and mitigate XML-based vulnerabilities in such systems.

This paper focuses on testing for XML Injections (*XMLi*), a prominent family of attacks that aim at manipulating XML documents or messages to compromise XML-based applications. Moreover, we target the front-end web applications of SOA systems, i.e., front-end web applications are the systems under test (SUTs) in our context. Among other functionalities, they receive user inputs, produce XML messages, and send them to services for processing (e.g., as part of communications with SOAP and RESTful web services [5], [6]). If such user inputs are not properly validated, *XMLi* attacks will likely occur. As a consequence, malicious XML messages, which are produced by front-end web applications resulting from *XMLi* attacks, can compromise services that consume these messages.

In practice, there exist approaches based on fuzz testing, e.g., ReadyAPI [7], WSFuzzer [8], that try to send some XML *meta-characters* (e.g., $<$) and seek for abnormal responses from the SUTs. These approaches might be able to detect simple *XMLi* vulnerabilities when the following two conditions are satisfied: (i) there is no mechanism in place to check XML well-formedness and validity, and (ii) erroneous responses of the SUTs are observable by the testing tools. However, they will typically fail to detect subtler vulnerabilities.

In this paper, we propose an automated approach based on a genetic algorithm to search for sophisticated and effective test cases (attacks) to detect *XMLi* vulnerabilities. Given the SUT, a web application that communicates with web services through XML messages, we first identify a set of possible malicious XML messages (called *test objectives* in this paper) that the SUT can produce and send to those services. This process can be fully automated based on known XML attacks and the XML schemas of those web services. Then, we use a genetic algorithm to search for inputs for the SUT (e.g., text data in HTML input forms) in an attempt to generate XML messages matching such test objectives (TOs, for short). Our search-based technique is guided by an objective function that measures the difference between the actual SUT outputs (i.e., the XML messages toward the web services) and the TOs. It does not require access to the source code of the SUT and can, thus, be applied in a black-box fashion on many different systems. The current paper focuses on the generation of test inputs and is complementary to the automated solution for generating TOs that we proposed in previous work [9].

Note that proper input validation in the front-end can prevent many of the possible security attacks. However, in the context of large web applications with hundreds of distinct input forms, some input fields are typically not properly validated as a result of time pressures, changes, and lack of security expertise.

Furthermore, some attacks could be based on the combination of more than one input field, where each field in isolation could pass the validation filter unaltered. In some cases, full data validation (i.e., rejection/removal of all potentially dangerous characters) is not possible, as meta-characters like $<$ could be valid, and ad-hoc solutions need to be implemented (which could be faulty). For example, if a form is used to

978-1-5090-6031-3/17 $31.00 © 2017 IEEE
DOI 10.1109/ICST.2017.39

input the message of a user, emoticons like <3 representing a "heart" can be quite common.

In this paper, we have carried out an extensive evaluation of the proposed approach on two case studies. The first study consists of 20 experiments on six web applications that simulate bank interactions with an industrial bank card processing service. These web applications have different levels of complexity in terms of the number of inputs, their data types and the validation technique. The second study includes a third-party application used for training purposes and an industrial web application having millions of registered users, with hundreds of thousands of visits per day. Results are promising, as our proposed search-based testing approach is effective at detecting *XMLi* vulnerabilities in both case studies, within practical execution time. The evaluation of our approach on such diverse systems, including a large industrial web application, is a sizable and useful empirical contribution.

The remainder of the paper is structured as follows. Section II provides background information on *XMLi* attacks, and describes the testing context of our research. Section III describes our proposed approach and the tool that we developed for its evaluation. Section IV reports and discusses our evaluation on two case studies including research questions, results and discussions. Section V discusses related work. Finally, Section VI concludes the paper.

II. Background

A. XML Injection

We briefly discuss *XMLi* attacks and illustrate them using a concrete example. We refer the reader to our previous work [9] for a more comprehensive categorization of *XMLi* attacks. XML Injection is an attack technique that aims at manipulating the logic of XML-based applications or services. It is carried out by injecting malicious content via XML tags and elements into input parameters to manipulate the XML messages that the system produces, e.g., to create malformed XML messages to crash a target system. XML Injection is also used to carry nested attacks (malicious content embedded in XML messages), e.g., the payloads for SQL Injection or cross-site scripting. The aim of this type of attack is to compromise the system itself or other systems that process the malicious XML messages, e.g, a back-end database that returns confidential information based on queries in the XML messages.

Consider an example in which users can register themselves through a web portal to a central service[1]. Once registered, a user can access different functionalities offered by the service. User registration data are stored in the XML registration database depicted in Figure 1. Notice that inside the database, each *user* element has a single child element, called *userid*, that is inserted by the application to assign privileges and which users are not allowed to modify.

The web portal has a web form with three user input fields: *username*, *password*, and *email*. Each time a user submits a

[1]This example is inspired by the example given by the Open Web Application Security Project (OWASP) https://www.owasp.org/index.php/Testing_for_XML_Injection_(OTG-INPVAL-008)

```
<?xml version="1.0" encoding="UTF-8"?>
  users>
  <user>
   <username>David</username>
   <password>Lux_230</password>
   <userid>300</userid>
   <mail>david@uni.lu</mail>
  </user>
  <user>
   <username>Mike</username>
   <password>spl3M3n</password>
   <userid>312</userid>
   <mail>mike@uni.lu</mail>
  </user>
  ...
  ...
</users>
```

Fig. 1. An example of an XML database for storing user registration.

registration request, the application invokes the following piece of Java code to create a XML SOAP message and sends it to the central service (which is a SOAP web service in this case). Notice that the *getNewUserId()* method is invoked to create a new user identifier and no user modification of *userid* is expected.

```
1 soapMessage = "<soap:Envelope><soap:Body>"
2 + "<user>"
3  + "<username>"+r.getParameter("username")+"
       </username>"
4  + "<password>"+r.getParameter("password")+"
       </password>"
5  + "<userid>"+getNewUserId() + "</userid>"
6  + "<mail>"+r.getParameter("mail")+"</mail>"
7 + "</user>"
8 + "</soap:Body></soap:Envelope>";
9 validate(soapMessage);
```

Even though there is a validation procedure at line 9, the piece of code remains vulnerable to XML injection attacks because user inputs are concatenated directly into the variable *soapMessage* without validation. Let us consider the following malicious inputs:

Username = `Tom`
Password = `Un6Rkb!e</password><!--`
E-mail = `--><userid>0</userid><mail>admin@uni.lu`

These inputs result in the XML message in Figure 2. The *userid* element is replaced with a new element having the value of "0", which we assume is reserved to the Administrator. In this way, the malicious user *Tom* can gain administration privilege to access all functionalities of the central service. The message is well-formed and valid according to the associated XML schema (i.e., the XSD). Therefore, the validation procedure does not mitigate this vulnerability.

Similarly, by manipulating XML to exploit *XMLi* vulnerabilities, attackers can inject malicious content that can carry other types of attacks. For instance, they can replace the value "0" above with "0 OR 1=1" for an SQLi attack. If the application directly concatenates the received parameter values into a SQL Select query, the resulting query is malicious and can result in the disclosure of confidential information when executed:

```
Select * from Users where userid = 0 OR 1=1
```

```
<soapenv:Envelope xmlns:soapenv="http://schemas.
    xmlsoap.org/soap/envelope/">
    <soapenv:Header/>
    <soapenv:Body>
        <user>
            <username>Tom</username>
            <password>Un6Rkb!e</password>
            <!--
                </password>
                <userid>500</userid>
                <mail>
            -->
            <userid>0</userid>
            <mail>admin@uni.lu</mail>
        </user>
    </soapenv:Body>
</soapenv:Envelope>
```

Fig. 2. An example of an injected SOAP message.

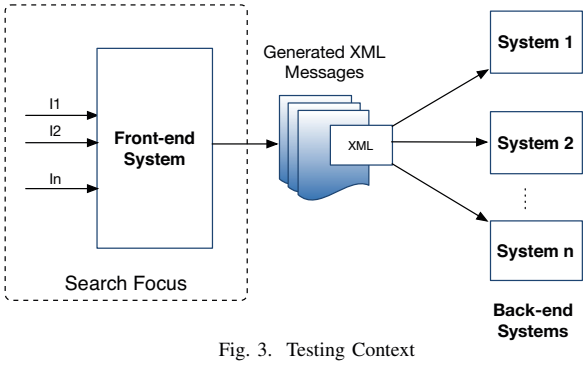

Fig. 3. Testing Context

B. Testing Context

A SOA system typically consists of a front-end web application that generates XML messages (e.g., toward SOAP and RESTful web services) upon incoming user inputs (as depicted in Figure 3). The front-end system often performs various transformation techniques on the user inputs before generating the XML messages, e.g., encoding, validation or sanitisation. XML messages are consumed by various back-end systems or services, e.g., an SQL back-end, that are not directly accessible from the net. In this paper, we focus on the front-end web application and aim to test if it is vulnerable to *XMLi* attacks. We consider the web application as a black-box. This makes our approach independent from the source code and the language in which it is written (e.g., Java, .Net, Node.js or PHP). Furthermore, this also helps broaden the applicability of our approach to systems in which source code is not easily available to the testers (e.g., external penetration testing teams). However, we assume to be able to observe the output XML messages produced by the SUT upon user inputs. To satisfy this assumption, it is enough to set up a proxy to capture network traffic leaving from the SUT, and this is relatively easy in practice.

The security of the front-end plays a vital role in the overall system's security as it directly interacts with the user. Consider, for instance, a point of sale (POS) as the front-end that creates and forwards XML messages to the bank card processors (bank-end). If the POS system is vulnerable to *XMLi* attacks, it may produce and deliver manipulated XML messages to web services of the bank card processors. Depending on how the service components process the received XML messages, their security can be compromised, leading to data breaches or services being unavailable, for example.

III. Approach

This section describes our search-based testing [10] approach to detect *XMLi* vulnerabilities. We first describe the TOs that are used to guide the search for malicious test inputs and demonstrate the presence of vulnerabilities. A search-based technique that generates inputs to reach such TOs is then introduced. Along with the discussion of the technique, we describe in detail its building blocks, including input encoding and the fitness function.

A. Test Objectives (TOs)

In our approach, TOs are specific outputs of the SUT (i.e., XML messages to web services) that contain malicious content. If there exist inputs (e.g., forms in HTML pages) that can lead the SUT to generate such malicious XML outputs, then the SUT is considered to be vulnerable.

A TO is said to be covered if we can provide inputs which result into the SUT producing the TO. Our focus of this paper is to search for such user inputs. Since the TO is malicious by design, the SUT is not expected to produce it and coverage indicates vulnerability. Sending such TOs to the backend systems/services could severely impact them depending on the malicious content that these TOs carry.

We define four types of TOs based on the types of *XMLi* attacks described in our previous work [9]: *Type 1: Deforming*, *Type 2: Random closing tags*, *Type 3: Replicating* and *Type 4: Replacing*. The intent and impact of each of these *XMLi* attacks types are different. *Type 1* attacks aim to create malformed XML messages to crash the system that process them. *Type 2* attacks aim to create malicious XML messages with an extra closing tag to reveal the hidden information about the structure of XML documents or database. Finally, *Type 3* and *Type 4* aim at changing the XML message content to embed nested attacks, e.g., SQL Injection or Privilege Escalation.

To obtain TOs for a given SUT, we first sample diverse and non-malicious output XML messages (recorded during normal execution of the SUT). Then, we automatically modify those XML messages to inject malicious intent in them. In our previous work [9], we have developed an approach and tool to automate TO generation.

Ensuring diversity in the generated TO set is important to the success of our technique. Since we consider the SUT as a black box, we do not know a priori how inputs are related to output XML messages. Having a diverse set of TOs increases our chance of figuring out such relationships and detecting *XMLi* vulnerabilities. Therefore, when generating TOs, we make sure that each XML element/attribute of the messages is modified at least once with all the types of *XMLi* attacks described in our previous work [9].

Consider the example of user registration given in Section II-A: Figure 2 shows a possible TO where the *userid*

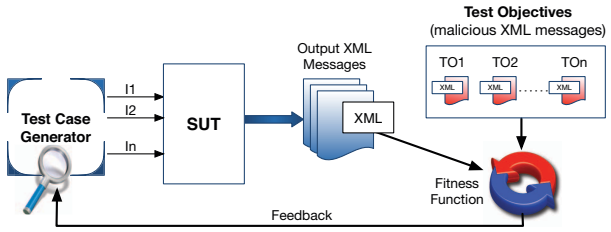

Fig. 4. The overall search-based approach to generating tests for reaching the TOs.

element is manipulated, i.e., the original element *userid* has a value 500, which has been commented out and replaced with the new *userid* element having a value of 0.

B. Search-based Testing

Once a set of TOs for the SUT is generated, we want to test whether the SUT, with certain user inputs, can produce any of these TOs. If the SUT does not properly block malicious inputs that lead to any of the TOs, it is considered vulnerable. The testing problem is now defined as a search problem: seeking for malicious input values (i.e., *XMLi* attack strings) that, when submitted to the the SUT, lead the SUT to produce XML messages matching the TOs.

As the number of possible input strings for the SUT can be extremely large, random testing without any guidance would most likely be ineffective. Also, since we consider the SUT as a black-box and hence we are not aware of how user inputs are transformed into output XML messages, it is infeasible to use a simple deterministic algorithm based on direct input-output matching. Therefore, we use a search algorithm, namely a genetic algorithm (GA) [11], that seeks to evolve the output of the SUT towards the TOs. Given a TO and an output XML message, the objective (fitness) function guiding search is measured by the *distance* between them.

GA is inspired by the mechanisms of natural adaptation. In a nutshell, it starts from an initial population of individuals (encoded in *chromosomes*) and iteratively evolves them by selecting elite individuals (having the best fitness) and producing offspring by applying crossover and mutation operators.

Our GA-based approach is depicted in Figure 4. Each test case is a GA individual, composed of a number of string values to be assigned to input parameters of the SUT. The number of input strings depend on the number of inputs of the SUT. For each TO, our approach first generates an initial population of random test cases. Then, it iteratively selects and evolves them based on the feedback from the fitness function. This process is repeated until optimal fitness is achieved (outputs match a TO) or we run out of computational budget (e.g., timeout). Note that we consider each TO separately. TOs are independently created based on different types of *XMLi* attacks. The coverage of different TOs requires different test cases and, hence, we cannot consider the coverage of multiple TOs at the same time.

In the subsequent sections we discuss in detail how test cases are encoded and the way we define and measure the fitness function.

1) GA Encoding and Reproduction: The encoding of chromosomes is a pivotal task when addressing a search problem with a GA. For a given SUT with N input parameters, we represent a test as a chromosome of N genes. Each of these genes is one string corresponding to an input parameter. We consider every input of the SUT as a string regardless of its original data type, since (i) inputs to the web application SUT will be sent through HTTP and (ii) malicious *XMLi* attacks are strings. These input strings can contain alphanumeric and special characters (e.g., %, || or &).

By default, we consider all the printable ASCII characters as the alphabet for the input strings. However, depending on the given TOs, we can reduce the alphabet to containing only characters that can possibly compose the TOs. We investigate this point further in Section III-B3.

Each input string is represented as an array of ASCII characters. The lengths of the arrays for the genes are set based on the expected maximum lengths of the corresponding input parameters. We use a special symbol to denote the "empty" character, i.e., absence of character, to fill up the array when the input string is shorter than the maximum length. Furthermore, in this way the lengths of input strings can vary during the search, as these empty spaces can be filled with new characters.

Reproduction to create offspring is done through standard crossover and mutation operators. Crossover is a genetic operator that is used to vary the programming of chromosome pairs from one generation to the next: a pair of individuals is selected, the chromosome of each is cut into two parts at the same random cut point, two new offspring are formed by concatenating the head part of one parent with the tail part of the other parent. Mutation on an offspring is done by randomly selecting a position in the chromosome and swap its corresponding character with a new one that is randomly selected from the alphabet.

To select chromosomes for crossover and mutation, we use the binary tournament selection technique [12] as recommended in the literature [13], [14].

2) Fitness Function: The fitness (objective) function that guides our search for effective test inputs is defined as the distance between the output XML message and the TO. Given a TO and an individual (i.e., a test case consisting of inputs to the SUT), its fitness is measured as the distance between the XML message that the SUT produces upon the execution of the test case and the TO. Our search evolves such individuals, aiming at reducing the distance so that the TO can be reached.

A TO is essentially a malicious XML document that might be malformed or contain potentially dangerous values (e.g., SQL injection). The structure of the TO might, therefore, be broken. As a result, we consider TOs as plain text, i.e. just as strings. We use string edit distance (also known as Levenshtein distance) between pairs of strings for the fitness measure. In short, edit distance between two strings is the minimum number of editing operations (inserting, deleting, or substituting a character) required to transform one into the

other. It is also supported by fast algorithms [15] and open source implementations.

Consider the following simple example of a TO of a SUT that has a single input parameter I. The TO is represented as the string:

```
<test>data OR 1=1</test>
```

Upon a test case t in which I = *" OR %"*, the SUT generates the following XML message:

```
<test>data OR %</test>
```

The fitness value of t is measured as the edit distance between the TO and the XML message, which in this case is equal to three, as we need to modify the "%" character into "1", and then add the two characters "=1".

The lower the fitness value is, the closer we are to cover a TO. We consider the TO to be covered when the corresponding fitness value is 0. It means that the generated XML message and the TO are identical.

3) Reducing the Search Space: The search space for a SUT is characterized by three factors: the number of input parameters of the SUT, the maximum string lengths for the values of those inputs, and the alphabet from which strings are created. They should be controlled in order to reduce the search space and improve the performance of the GA. The number of input parameters of the SUT is normally fixed. However, assuming that some domain knowledge is available i.e., if we know that some parameters are not involved in producing XML outputs, then they can be excluded from the search. Their values can be fixed with valid data taken from the functional test suite of the SUT.

The maximum string length values, i.e., the size of the genes, corresponding to the input parameters of the SUT, should also be adapted to the nature of the parameters. For instance, parameters for *name* and *password* are often shorter than a parameter for *description*. Again, if we know any upper bound for input lengths, we should limit the gene size accordingly. Nevertheless, since the input strings are used or concatenated to create XML messages, their lengths are smaller than the lengths of the XML messages. As a result, we should always set the string lengths smaller than the lengths of TOs.

We should also consider to restrict the alphabet to contain only the specific set of characters that appear in the TOs. If the TOs do not contain some characters, we omit them from the alphabet. As the GA has to work with fewer characters in this case, we expect an improvement in performance. We investigate the benefit of limiting the alphabet in the experimental evaluation.

C. Tool Implementation

We developed a tool in Java that implements the technique presented in this paper. The inputs of the tool include the TOs that contain malicious intents for a target SUT. The tool generates test cases, which include the values for the input parameters of the SUT (e.g., input values in HTML forms). When these tests are run, the tool compares the XML outputs of the SUT to the TOs for fitness calculation. The search is guided by this fitness function for generating new test cases. The process is repeated until the TOs are covered or the tool runs out of time.

The main components of the tool are: Test Case Generator and Test Executor. The test case generator is the core component of the tool. It is implemented on top of jMetal [16] - an object oriented, Java-based framework for optimization. The test executor provides an interface between the SUT and the test case generator. It takes the input values generated by the test case generator and submits them to the SUT (e.g, through a HTTP POST with all the needed cookies set up). The XML messages sent by the SUT toward the web services need to be intercepted and then sent back to the test case generator where the fitness is calculated. This can be easily done by setting up an HTTP proxy between the SUT and the web services.

The test case generator is generic and can be used with any application. The test executor has been modularized so that it can easily be instantiated for a specific SUT that has a different user interface (e.g., HTML web forms with different parameter names) and that generates XML files in different ways (e.g., SOAP messages or data bodies in HTTP POST messages toward RESTful web services).

IV. Evaluation

We evaluate the effectiveness in vulnerability detection and the execution cost of the proposed approach through a series of experiments grouped into two studies, namely *Study 1: Controlled Experiments* and *Study 2: Third-party Applications*. The common characteristic of the subject applications in these studies is that they receive user inputs, produce XML messages, and process or send them to associated web services. In Study 1, we have two front-ends, called *SBank* and *SecureSBank*, that simulate a bank's interactions with a real-world bank card processing service[2]. Since we designed these front-ends, we can control the number of inputs and validation mechanisms in them to investigate how they influence the effectiveness of our approach.

Differently from Study 1, Study 2 involves third-party independent subject applications. They enable us to evaluate how well our approach scales and to which extent our results generalize.

A. Research Questions

In this paper, we investigate the following six research questions:

RQ1 [*Effectiveness*]: *To what extent is our search-based approach effective in detecting* XMLi *vulnerabilities?*

Since our TOs correspond to malicious XML messages, being able to identify inputs to generate them would demonstrate that the SUTs are vulnerable. Our approach is therefore deemed effective if it can find input strings that lead to the production of TOs, i.e, cover TOs.

RQ2 [*Comparison with Random Search*]: *How does our search-based approach perform compared to random search?*

[2]The name of the company cannot be revealed due to non-disclosure agreements.

Random search [12] is typically adopted as a baseline in SBSE research [17].

RQ3 [*Cost*]: *What is the cost, in terms of execution time, of applying the proposed approach?*

We should consider the cost for deriving TOs and the computational cost of the GA. Since the TO derivation is addressed in our previous work [9] and is not the focus of the current paper, we only discuss here the input search cost. As GAs are notorious for being computationally expensive, we investigate whether the cost in terms of execution time affects the applicability of our approach.

RQ4 [*Impact of Restricted Alphabets*]: *How does restricting the alphabet to the characters in the TO affect the GA performance?*

As described in Section III-B3, instead of using the complete alphabet of all possible characters, we could use a restricted alphabet by omitting the characters unused in the TOs. It is expected that the performance of the GA would improve as the overall search space would be reduced. We assess whether the magnitude of the improvement is practically significant.

RQ5 [*Influence of Input Setting*]: *To which degree the number of input parameters and their length settings affect the GA performance?*

When there are more user inputs, the GA has to spend more time searching. Also, the length of user inputs can be variable (Var) and depend on specific inputs, or be fixed (Fix) for all inputs. The former may help reduce the search space, but it requires selecting proper lengths for inputs, which often requires domain knowledge. The latter is easier as one has to select a single maximum length for all inputs, but it might unnecessarily increase the search space. We study these trade-offs in **RQ5**.

RQ6 [*Influence of Input Validation*]: *Does search-based testing work in presence of input validation?*

The SUT may implement protection mechanisms that do validate the inputs. When such mechanisms detect malicious inputs, they often react to malicious content by generating error messages in the HTTP responses, and often end up not generating any XML to be sent toward the target web services. This is a challenge for the GA, as no useful fitness value would be produced, generating a fitness plateau hampering effective search. It is hence important to investigate whether our proposed approach works in such circumstances, i.e., how it is affected by input validation.

B. Metrics

We rely on the following metrics to help answer the above research questions.

- Coverage Rate (C): Coverage rate C is the ratio of the number of covered TOs over the total number of TOs. We denote the coverage rates for the GA as C_{ga} and for random search as C_{rn}.
- Average Execution Time (T): Each experiment on a specific TO is repeated 10 times to account for randomness. We then measure the average execution time T (minutes) across runs and per TO for specific subject applications. We denote the execution time as T_{ga} and T_{rn} for the GA and random search, respectively.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.
org/soap/envelope/" xmlns:lu="[redacted]">
    <soapenv:Header/>
    <soapenv:Body>
        <lu:perform>
            <lu:resInput>
                <lu:UserName>Tom</lu:UserName>
                <lu:IssuerBankCode>0231</lu:IssuerBankCode>
                <lu:RequestId>28278111TOM</lu:RequestId>
                <lu:CardNumber>123456789</lu:CardNumber>
            </lu:resInput>
        </lu:perform>
    </soapenv:Body>
</soapenv:Envelope>
```

Fig. 5. An example of output XML message created by *SBank*.

C. Study 1: Controlled Experiments

Beside checking the effectiveness of the proposed approach on applications that are known to be vulnerable, this study investigates the influence of the input space (number of inputs, their lengths, and the alphabet) on the GA performance.

1) Subject Applications: Our first study has been carried out with two web applications, *SBank* and *SecureSBank*, that simulate the web front-end of a banking system that receives user inputs, produces XML messages, and sends them to a bank card processing service. The XML messages share the same structure as those used in production. *SBank* and *SecureSBank* were, however, developed in our lab specifically for this study. We wanted to be able to configure the number of user input parameters and apply validation mechanisms. *SBank* and *SecureSBank* were deliberately designed to be vulnerable.

Both applications can have up to three input parameters, including *UserName*, *IssuerBankCode*, and *CardNumber*. The applications generate XML messages using the inputs submitted by the user. Figure 5 shows an example of such an output XML message. Note that the *RequestID* element in the XML message is generated by the application automatically. Users are not authorized to tamper with its value unless they do so maliciously. The generated XML messages are then forwarded to the web services of the card processing company.

SBank directly inserts the user inputs into the XML elements of the message without validation. This makes the application vulnerable to *XMLi* attacks. *SecureSBank* is similar to *SBank* except that one of the input parameters is validated. The application validates the input parameter *IssuerBankCode* and generates an error message if malicious content is found. The aim of evaluating *SecureSBank* is to test our approach in the presence of input validation.

2) Test Objectives: As described in Section III-A, we created TOs based on the four *XMLi* injection techniques proposed in our previous work [9]. For *SBank* and *SecureSBank*, TOs are created by applying these four *XMLi* techniques to every element of the sample XML message (Figure 5). The *Type 4: Replacing* attack is a more advanced form of *XMLi* that requires at least two XML elements where the value of one must be auto-generated by the application. Therefore this attack can only be applied to the *RequestID* as it is

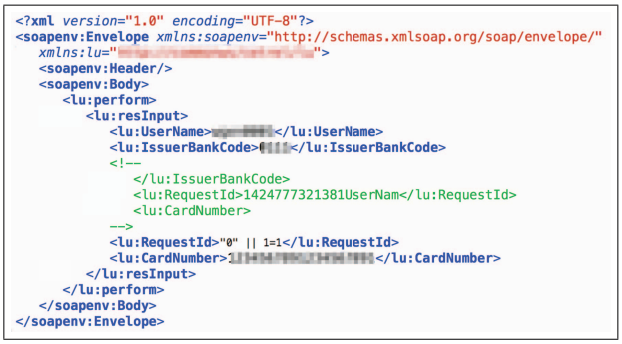

Fig. 6. An example of test objective containing a malicious attack.

the only element auto-generated by *SBank*/*SecureSBank*. This results in 10 (3 attacks x 3 elements + 1 attack x 1 element) representative TOs in total.

An example of a TO for *SBank* is shown in Figure 6. It contains four XML elements *UserName, IssuerBankCode, RequestId*, and *CardNumber*. The value of the parameter *RequestID* includes malicious content ("0" $||$ $1 = 1$ - an embedded SQLi attack). The application of the *XMLi* technique *Type 4: Replacing* resulted in commenting out the previous element and inserting the new *RequestID* element along with malicious content. If the web service that consumes such malicious TO executes a SQL query by directly concatenating the received values into it, then the resulting query may get executed:

```
Select * from Cards where RequestID = "0" || 1=1
```

The condition in this SQL query is a tautology and results in returning all cards' information when executed. If the front-end system can generate this XML message from user inputs, it is considered vulnerable to *XMLi* attacks.

3) Experiment Settings: We conducted a number of experiments with different settings of the applications, as represented in Table I. Each row in the table represents one experiment. The column *Exp. ID* is the id of the experiments. The values of *Exp. ID* is based on the corresponding applications (S for *SBank*, SS for *SecureSBank*), the number of inputs considered, whether input lengths are fixed (Fix) for all or are input specific (Var), and whether the alphabet is restricted (Y) or not (N). The *TOs* column lists the number of TOs created for the experiments. The *#Inp.* column lists the number of inputs accepted by the application version. The *Len.* column represents whether the length of the genes in a chromosome is fixed or variable. The gene's lengths directly correspond to the lengths of the input parameters of the SUT and may affect the GA's performance. The last column *Res. Alph.* indicates whether the GA uses a reduced alphabet or not.

All but four experiments have 10 TOs, as described in Section IV-C2. Each of these four experiments (i.e. S.1.* and SS.1.*) have 9 TOs as they consist of the *SBank*/*SecureSBank* versions having one input parameter. The missing TO from each of these four experiments is the one generated using the *Type4: Replacing* attack on *RequestID* element as this TO requires at least two input parameters to be covered. It is therefore not coverable in the experiments with application versions having only one input parameter.

TABLE I
EXPERIMENT SETTINGS FOR STUDY 1: SSBANK STANDS FOR *SecureSBank*, EXPERIMENT ID (EXP. ID) IS NAMED BASED ON THE CORRESPONDING APPLICATION (APP.), THE NUMBER OF INPUTS (#INP.), THE INPUT LENGTH SETTING (LEN), AND WHETHER THE ALPHABET IS RESTRICTED (RES. ALPH.).

App.	Exp. ID	#TOs	#Inp.	Len.	Res. Alph.
SBank	S.1.F.N	9	1	Fix	N
	S.1.F.Y	9	1	Fix	Y
	S.2.F.N	10	2	Fix	N
	S.2.F.Y	10	2	Fix	Y
	S.2.V.N	10	2	Var	N
	S.2.V.Y	10	2	Var	Y
	S.3.F.N	10	3	Fix	N
	S.3.F.Y	10	3	Fix	Y
	S.3.V.N	10	3	Var	N
	S.3.V.Y	10	3	Var	Y
SSBank	SS.1.F.N	9	1	Fix	N
	SS.1.F.Y	9	1	Fix	Y
	SS.2.F.N	10	2	Fix	N
	SS.2.F.Y	10	2	Fix	Y
	SS.2.V.N	10	2	Var	N
	SS.2.V.Y	10	2	Var	Y
	SS.3.F.N	10	3	Fix	N
	SS.3.F.Y	10	3	Fix	Y
	SS.3.V.N	10	3	Var	N
	SS.3.V.Y	10	3	Var	Y

For these experiments, a TO is covered when it matches the output XML message, meaning that fitness is 0. As GA is a randomized algorithm, to account for the statistical variation, we run it 10 times for each TO. To simplify the discussion and visualization of the experiment results, a TO is considered to be covered if the TO is covered in at least one of these 10 runs. Due to the large number of experiments and space constraints, we cannot report in detail the results on each of the TO.

For each iteration of the TO, the program is terminated when the TO is covered or when there is no improvement in fitness after X evaluations. The value of X is determined based on some preliminary experiments. We use the same termination criteria for the search-based approach and random search.

The GA algorithm described in Section III-B is generic. More specifically, we use generational GA [12], which is one of the single-objective optimization algorithms available in jmetal [16] and has worked well for our problem based on our preliminary experiments. The GA parameters have also been assigned values based on some small-scale preliminary experiments and are consistent with the "best practices" in the literature [16], [18]. The population size is set to 50, which is in the recommended range of 30-80 [19]. The crossover rate is set to 0.8, which also falls within the recommended range of 0.45 to 0.95 [20], [19]. For the mutation rate, we use the recommendations in [20], [21] i.e. $\frac{1.75}{\lambda\sqrt{l}}$ where l is the length of the chromosome and λ is the population size.

4) Results: The results obtained with Study 1 are shown in Tables II and III for *SBank* and *SecureSBank*, respectively. Our proposed approach achieved 100% TO coverage for the experiments with *SBank*. This was the case for all variants of *SBank* with different numbers of inputs (S.1.*, S.2.*, or S.3.*) and the GA configurations ([F|V].[Y|N]). These results

TABLE II
RESULTS FOR *SBank* FOR THE GA AND RANDOM SEARCH IN TERMS OF COVERAGE RATE C_{ga}, C_{rn} AND AVERAGE EXECUTION TIME T_{ga}, T_{rn} IN MINUTES.

Exp.ID	C_{ga}	C_{rn}	T_{ga}	T_{rn}
S.1.F.N	9/9	0/9	13.63	54.75
S.1.F.Y	9/9	0/9	10.53	54.92
S.2.F.N	10/10	0/10	28.30	49.60
S.2.F.Y	10/10	0/10	23.57	50.88
S.2.V.N	10/10	0/10	16.78	42.30
S.2.V.Y	10/10	0/10	12.18	45.48
S.3.F.N	10/10	0/10	31.88	40.42
S.3.F.Y	10/10	0/10	25.20	43.78
S.3.V.N	10/10	0/10	24.13	42.28
S.3.V.Y	10/10	0/10	19.13	47.93

TABLE III
RESULTS FOR *SecureSBank* FOR THE GA AND RANDOM SEARCH IN TERMS OF COVERAGE RATE C_{ga}, C_{rn} AND AVERAGE EXECUTION TIME T_{ga}, T_{rn} IN MINUTES.

Exp.ID	C_{ga}	C_{rn}	T_{ga}	T_{rn}
SS.1.F.N	6/9	0/9	14.32	52.07
SS.1.F.Y	6/9	0/9	11.70	44.82
SS.2.F.N	0/10	0/10	25.83	38.28
SS.2.F.Y	0/10	0/10	24.20	31.63
SS.2.V.N	6/10	0/10	24.60	40.72
SS.2.V.Y	6/10	0/10	19.62	36.05
SS.3.F.N	0/10	0/10	18.20	27.88
SS.3.F.Y	0/10	0/10	17.95	23.63
SS.3.V.N	6/10	0/10	23.48	36.98
SS.3.V.Y	6/10	0/10	21.05	38.85

demonstrate that all the variants of *SBank* are found to be vulnerable to *XMLi*.

Results for *SecureSBank* are provided in Table III. They also show that all the variants of *SecureSBank*, with different number of user inputs (SS.1.*, SS.2.*, or SS.3.*), have at least one configuration of the GA ([F|V].[Y|N]) that covers more than half the TOs, meaning that the variants are found to be vulnerable. However, the coverage rate is smaller than that achieved with *SBank*. This was expected, as with the presence of input validation procedures applied on the input parameter *BankCode*, four (three for SS.1.*) TOs that require malicious inputs for this parameter are not feasible. For the other TOs, it is also more difficult for the GA to search since, whenever an unexpected input is submitted to *BankCode*, *SecureSBank* produces an error message. Since it differs from the TOs by a large distance, it introduces large fluctuations in the fitness function over time during the search.

The average execution time T_{ga} per TO ranges from approximately 10 to 31 minutes for *SBank* and 11 to 25 minutes for *SecureSBank*. Such durations in practice are acceptable as they are not expected to impact the pace of development, especially when testing is fully automated and performed offline, for example on continuous integration systems.

Regarding **RQ2**, random search could not cover a single TO in any configuration as depicted in the C_{rn} columns in both Tables II and III. This clearly shows that our search problem is far from trivial. Even with just 10 runs, Fisher exact tests on differences between success rates C_{rn} and C_{ga} provide very small p-values close to 0. Also the execution time T_{rn} for random search in each experiment is much higher than their counterparts in the GA experiments.

Regarding the research questions **RQ1**-**RQ3**, we find that:

> *The proposed approach is highly effective in searching for inputs to detect* XMLi *vulnerabilities and performs much better than random search. The average execution time per TO is acceptable in practice.*

All of the experiments that used a restricted alphabet (Exp. IDs ending with Y) performed better in terms of execution time. For instance, the average execution time of S.3.F.Y (which used a restricted alphabet) per TO is 25 minutes whereas its counterpart S.3.F.N (which used the full alphabet) stands around 31 minutes. For **RQ4**, we find that:

> *Using a restricted alphabet for the GA helps significantly reduce the execution time.*

The effects of varying the number of user inputs on the GA performance are evident from the result tables of both subject applications. The application variants with one input parameter (S.1.*, SS.1.*) achieved the best results in terms of execution time. The variants with two input parameters performed better compared to their three-parameter counterparts. For example, S.2.* achieved the same TO coverage in less time compared to S.3.*.

The results of both applications indicate that keeping the input length fixed is not a viable option. The experiments where the lengths of the input parameter's lengths were variable (specific to parameters) achieved much lower execution times compared to their fixed length counterparts. For example, S.3.V.* with variable length took less time than its counterparts S.3.F.*. The results of *SecureSBank* comparing length options are even more interesting, as none of the configurations with two and three input parameters with fixed length (S.2.F.*, S.3.F.*) could cover any TO. In contrast, the variable length configurations performed much better. For instance, SS.2.V.* with variable length achieved 6/10 coverage while its counterpart SS.2.F.* with fixed length did not cover any TO.

Despite such observations, it is important to note that the differences between variable and fixed length options for the input parameters depend on the variation in their actual lengths. If there exists a significant variation in the actual lengths while the GA is configured to use the same length for all parameters, then the GA performance will be considerably affected. For instance, for *SBank* and *SecureSBank*, the length of the *CardNumber* parameter is 16 characters while the *BankCode* is only four characters. On the other hand, if the lengths of the parameters differ by only a few characters, then keeping it fixed (maximum length) for all parameters will not have much effect on the coverage or execution time. In short, in addressing **RQ5**, we find that:

> *The number of user input parameters affects the execution time. Furthermore, using a maximum and identical fixed length for all user inputs does adversely affect coverage as well as execution time.*

Regarding input validation for **RQ6**, all variants of the application *SecureSBank* achieved much lower coverage (Table III) compared to the non-validated *SBank* variants (Table II). However our approach still covered more than half of the TOs in total (excluding the four infeasible TOs) for *SecureSBank*.

D. Study 2: Third-party Applications

We focus exclusively on the most effective experimental settings identified in Study 1 for these applications. The results of these experiments provide additional evidence to answer **RQ1**, **RQ3** and **RQ4**, The other RQs were not further investigated due to space and technical constraints, as further described below.

1) Subject Applications: Two subject applications considered in this study are *XMLMao* and M (an arbitrary name to preserve confidentiality).

- **XMLMAO:** *XMLMao* is a deliberately vulnerable web application for testing XMLi vulnerabilities. It is part of the Magical Code Injection Rainbow (MCIR) [22] framework for building a configurable vulnerability test-bed. The application has a single user input parameter. It inserts inputs directly into one of the four locations in the output XML message, depending on *XMLMao*'s setting. *XMLMao* is written in PHP and consists of 1178 lines of code.
- **M:** M is an industrial web application with millions of registered users and hundreds of thousands of visits per day. The application itself is hundreds of thousands of lines long, communicating with several databases and more than 50 corporate web services (both SOAP and REST). Out of hundreds of different HTML pages served by M, in this paper we focus on one page having a form with two string inputs. Due to non-disclosure agreements and security concerns, no additional details can be provided on this case study.

2) Test Objectives: To create TOs for *XMLMao*, we apply the first three *XMLi* mutation techniques (Types $1-3$) to the sample XML message of the application. *Type 4: Replacing* is not applicable as it requires two user inputs. Since there are four locations of insertion in the XML message, we create 12 (3 x 4) TOs.

We only used four TOs for M (one per type), as the experiments had to be run on the laptop of one of the engineers working on M, as opposed to a cluster for other experiments. The TOs come from one of the SOAP web services invoked by the SUT when the target HTML form is submitted. As M is a large and complex system, when run on a laptop, instead of a high performance cluster of servers, response times are slow and this makes fitness evaluation more time consuming (most of the 32 minutes in Table V). Even if response times for a single user are still in the order of tens of milliseconds, this made running a large number of experiments on M impossible, thus resulting in considering four TOs only.

3) Experiment Settings: Experiment settings for Study 2 are depicted in Table IV. Like Study 1, each row in the table represents one experiment. For *XMLMao*, there is only one input and, therefore, the Fix/Var length settings are not applicable.

TABLE IV
EXPERIMENT SETTINGS FOR STUDY 1: EXPERIMENT ID (EXP. ID) IS NAMED BASED ON THE CORRESPONDING APPLICATION (APP.), THE NUMBER OF INPUTS (#INP.), THE INPUT LENGTH SETTING (LEN), AND WHETHER THE ALPHABET IS RESTRICTED (RES. ALPH.).

App.	Exp. ID	#TOs	#Inp.	Len.	Res. Alph.
XMLMao	X.1.F.N	12	1	Fix	N
	X.1.F.Y	12	1	Fix	Y
M	M.2.F.Y	4	2	Fix	Y

TABLE V
RESULTS FOR XMLMAO AND M IN TERMS OF COVERAGE RATE C_{ga} AND AVERAGE EXECUTION TIME T_{ga} IN MINUTES.

Exp. ID	C_{ga}	T_{ga}
X.1.F.N	12/12	6.86
X.1.F.Y	12/12	5.68
M.2.F.Y	1/4	31.87

The only two configurations for *XMLMao* are: X.1.F.Y with restricted alphabet and X.1.F.N without restricted alphabet for the GA. For *XMLMao*, we keep the same termination criteria as Study 1.

For M, there are two user inputs. However, due to computational constraints, we only considered one configuration (M.2.F.Y) with restricted alphabet. Furthermore, we used a 30K cap for the maximum number of fitness evaluations.

4) Results: The results obtained with Study 2 are shown in Table V. Our proposed approach achieved 100% TO coverage for *XMLMao*, i.e., all 12 TOs were covered. The cost in terms of execution time per TO is in the range 5-7 minutes.

With 31.87 minutes on average, execution time for M is much higher, although still within reasonable limits, but for a much lesser budget i.e. 30K evaluations. However, our technique did manage to find inputs to cover one of the TOs. The other three TOs turned out to be infeasible due to the type of input validation that M applies. Note that, in contrast to *XMLMao* that is a vulnerable application implemented to study and evaluate penetration testing tools, M is an actual industrial system used in production (i.e., the found potential vulnerability was not artificially injected for the sake of these experiments).

Regarding the research questions **RQ1** and **RQ3**, we find that:

> *The proposed approach is effective in finding inputs that detect XMLi vulnerabilities in third-party independent applications, within practical execution time.*

Regarding **RQ4**, the use of the restricted alphabet in XMLMao was also beneficial in terms of execution time, i.e., X.1.F.Y that used the restricted alphabet was faster with an average execution time of 5.68 minutes per TO compared to its counterpart X.1.F.N, with 6.86 minutes.

> *Using a restricted alphabet for the GA results in better execution time for XMLMao.*

V. RELATED WORK

In this section we survey work that targets XML vulnerabilities in web applications and services. We also consider existing

work that uses search-based test generation in security testing.

A. Testing for XML Vulnerabilities

Much of the research effort has been devoted to the detection of SQL injection and cross-site scripting vulnerabilities (e.g., [23], [24], [25]), whereas only limited research exists for XML-based vulnerabilities. A description of XML-based vulnerabilities and some possible countermeasures are given by Gupta and Thilagam [26]. In one of our previous work [27], we performed a comprehensive empirical study of two XML-based vulnerabilities, *Billion Laughs* (*BIL*) and *XML External Entity* (*XXE*), in modern XML parsers and found most of them vulnerable. There also exists some work on XML Denial of Services (XDoS) attacks [28], [29], [30]. Rosa et al. [31] have also studied *XMLi* attacks but in the context of intrusion detection, not security testing. None of this work addresses the issue of security testing of web services for *XMLi*. As web services are very common in large industrial systems (e.g., the one used in our case study in this paper) and *XMLi* entail serious risks, effective and automated testing focused on their detection is required.

In our previous work [9], we discussed four types of *XMLi* attacks and an approach based on input mutation for testing web services against them. Different from our current work in which we focus on front-ends of SOA systems that produce XML messages, in [9] we focused on the web services that consume XML messages. Our current work is therefore extending [9] to handle the cases when web services are not directly accessible from the internet i.e., only front-end systems can send service requests. Being the potential source of malicious XML messages to the back-end services, the security testing of front-end systems is very important.

Furthermore, in contrast to [9] where we used constraint solving and input mutation for manipulating XML messages, our approach in this paper is based on search-based testing techniques to generate test inputs that lead the SUT to produce malicious XML messages. Such inputs can then help detect *XMLi* vulnerabilities in web applications that can be exploited through their front-end.

B. Search-based Security Testing

Search-based software testing has mostly focused on functional testing [32], [33], [18] while non-functional aspects, and especially security testing, have received only limited attention [34], [35]. Avancini and Ceccato [36] have used search-based testing for detecting cross-site scripting vulnerabilities in web applications. They first use static analysis to detect candidate cross-site scripting vulnerabilities in PHP code. A genetic algorithm together with a constraint solver is then used to search for input values that can trigger the vulnerabilities. In contrast, our approach is a black-box testing technique that targets *XMLi* vulnerabilities.

Thomé et al. [37] also used a search-based technique for the security testing of web applications. Their approach systematically evolves inputs to expose SQL injection vulnerabilities by assessing the effects on SQL interactions between the web server and database. Our search-based testing approach also focuses on evolving test inputs but we address a different type of vulnerabilities, *XMLi* attacks. Moreover, Thomé et al. used a fitness function based on a number of factors that indicate the likelihood that the output is resulting from *SQLi* attacks. In contrast, we use a fitness function based on the distance between the SUT's outputs and automatically derived test objectives based on attack patterns.

There exist other vulnerability detection techniques [38], [39] that rely on evolutionary algorithms. Unlike our black-box approach for *XMLi* testing, these techniques are white-box and are used for buffer overflow detection.

VI. Conclusion

In this paper, we have presented an effective search-based approach for the security testing of web applications, with a focus on *XMLi* vulnerabilities. Our approach is able to lead the system under test (SUT) to produce malicious XML messages from user inputs (e.g., HTML forms). Such web applications often act as front-ends to the web services of a SOA system. In such context, *XMLi* vulnerabilities are common and can lead to severe consequences, e.g., DoS or data breaches. Therefore, automated and effective testing to detect and fix *XMLi* vulnerabilities is of paramount importance.

The proposed approach is divided into two steps: (1) the automated identification of malicious XML messages (our test objectives, TOs) that, if generated by the SUT and sent to services, would suggest a vulnerability; (2) The automated generation of SUT inputs that generate messages matching such TOs. This paper focuses on item (2), as item (1) was already addressed in our previous work [9].

In this paper, we have evaluated our novel approach on several artificial systems and one large industrial web application. Our results suggest that the proposed approach is effective as it was able to uncover vulnerabilities in all case studies. We also found that the employed genetic algorithm works best when some domain knowledge about the system under test is available, e.g., the lengths of user input parameters and their alphabets, to restrict the search space.

The proposed search-based testing approach is not only limited to XML Injection detection, but can be generalized to the detection of other types of vulnerabilities. For instance, to apply it to Cross-site scripting or SQL injection vulnerabilities, one only needs to modify the TOs according to the corresponding types of attacks for that vulnerability. Our future work will extend our current technique to cover more vulnerabilities.

VII. Acknowledgements

This work was supported by the National Research Fund, Luxembourg (grant FNR/P10/03 and FNR6024200) and the European Research Council under the European Unions Horizon 2020 research and innovation program (grant agreement number 694277). We are also grateful to our industry collaborators and Dennis Appelt for their valuable feedback in this paper.

REFERENCES

[1] M. N. Huhns and M. P. Singh, "Service-oriented computing: key concepts and principles," *IEEE Internet Computing*, vol. 9, no. 1, pp. 75–81, Jan 2005.

[2] "Extensible Markup Language (XML)," https://www.w3.org/XML/, accessed: 2016-04-26.

[3] "XML Vulnerabilities Introduction," http://resources.infosecinstitute.com/xml-vulnerabilities/, accessed: 2016-04-26.

[4] M. Jensen, N. Gruschka, and R. Herkenhöner, "A Survey of Attacks on Web Services," *Computer Science - Research and Development*, vol. 24, no. 4, pp. 185–197, 2009. [Online]. Available: http://dx.doi.org/10.1007/s00450-009-0092-6

[5] P. Adamczyk, P. H. Smith, R. E. Johnson, and M. Hafiz, *REST: From Research to Practice*. New York, NY: Springer New York, 2011, ch. REST and Web Services: In Theory and in Practice, pp. 35–57. [Online]. Available: http://dx.doi.org/10.1007/978-1-4419-8303-9_2

[6] "Simple Object Access Protocol (SOAP)," https://www.w3.org/TR/soap/, accessed: 2016-04-26.

[7] "SmartBear ReadyAPI," http://smartbear.com/product/ready-api/overview/, accessed: 2016-04-26.

[8] "WSFuzzer Tool," https://www.owasp.org/index.php/Category:OWASP_WSFuzzer_Project, accessed: 2016-04-26.

[9] S. Jan, C. D. Nguyen, and L. Briand, "Automated and Effective Testing of Web Services for XML Injection Attacks," in *Proceedings of the 2016 International Symposium on Software Testing and Analysis (ISSTA)*, Jul 2016.

[10] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test case generation," *Software Engineering, IEEE Transactions on*, vol. 36, no. 6, pp. 742–762, 2010.

[11] D. E. Goldberg *et al.*, *Genetic algorithms in search optimization and machine learning*. Addison-wesley Reading Menlo Park, 1989, vol. 412.

[12] S. Luke, *Essentials of Metaheuristics*, 2nd ed. Lulu, 2013, available for free at https://cs.gmu.edu/~sean/book/metaheuristics/.

[13] D. E. Goldberg and K. Deb, "A comparative analysis of selection schemes used in genetic algorithms," in *Foundations of Genetic Algorithms*. Morgan Kaufmann, 1991, pp. 69–93.

[14] J. Zhong, X. Hu, J. Zhang, and M. Gu, "Comparison of performance between different selection strategies on simple genetic algorithms," in *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06)*, vol. 2. IEEE, 2005, pp. 1115–1121.

[15] W. J. Masek and M. S. Paterson, "A faster algorithm computing string edit distances," *Journal of Computer and System sciences*, vol. 20, no. 1, pp. 18–31, 1980.

[16] J. J. Durillo and A. J. Nebro, "jMetal: A Java framework for multi-objective optimization," *Advances in Engineering Software*, vol. 42, pp. 760–771, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0965997811001219

[17] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Survey*, vol. 45, no. 1, pp. 11:1–11:61, Dec. 2012. [Online]. Available: http://doi.acm.org/10.1145/2379776.2379787

[18] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification & Reliability*, vol. 14, no. 2, pp. 105–156, Jun. 2004. [Online]. Available: http://dx.doi.org/10.1002/stvr.v14:2

[19] H. G. Cobb and J. J. Grefenstette, "Genetic algorithms for tracking changing environments," in *Proceedings of the 5th International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993, pp. 523–530. [Online]. Available: http://dl.acm.org/citation.cfm?id=645513.657576

[20] L. C. Briand, Y. Labiche, and M. Shousha, "Using genetic algorithms for early schedulability analysis and stress testing in real-time systems," *Genetic Programming and Evolvable Machines*, vol. 7, no. 2, pp. 145–170, 2006. [Online]. Available: http://dx.doi.org/10.1007/s10710-006-9003-9

[21] R. L. Haupt and S. E. Haupt, *Practical genetic algorithms*. John Wiley & Sons, 2004.

[22] "Magical Code Injection Rainbow (MCIR)," https://github.com/SpiderLabs/MCIR/, accessed: 2016-04-26.

[23] D. Appelt, C. D. Nguyen, and L. Briand, "Behind an Application Firewall, Are We Safe from SQL Injection Attacks?" in *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, April 2015, pp. 1–10.

[24] T. Gallagher, "Automated detection of cross site scripting vulnerabilities," March 2008, uS Patent 7,343,626. [Online]. Available: https://www.google.com/patents/US7343626

[25] M. Junjin, "An approach for sql injection vulnerability detection," in *Information Technology: New Generations, 2009. ITNG '09. Sixth International Conference on*, April 2009, pp. 1411–1414.

[26] A. N. Gupta and D. P. S. Thilagam, "Attacks on Web Services Need to Secure XML on Web," *Computer Science and Engineering, An International Journal (CSEIJ)*, vol. 3, no. 5, 2013. [Online]. Available: http://airccse.org/journal/cseij/papers/3513cseij01.pdf

[27] S. Jan, C. D. Nguyen, and L. Briand, "Known XML Vulnerabilities Are Still a Threat to Popular Parsers and Open Source Systems," in *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*, Aug 2015, pp. 233–241.

[28] A. Falkenberg, C. Mainka, J. Somorovsky, and J. Schwenk, "A New Approach towards DoS Penetration Testing on Web Services," in *Web Services (ICWS), 2013 IEEE 20th International Conference on*, June 2013, pp. 491–498.

[29] S. Suriadi, A. Clark, and D. Schmidt, "Validating Denial of Service Vulnerabilities in Web Services," in *Network and System Security (NSS), 2010 4th International Conference on*, Sept 2010, pp. 175–182.

[30] X. Ye, "Countering DDoS and XDoS Attacks against Web Services," in *Embedded and Ubiquitous Computing, 2008. EUC '08. IEEE/IFIP International Conference on*, vol. 1, Dec 2008, pp. 346–352.

[31] T. Rosa, A. Santin, and A. Malucelli, "Mitigating XML Injection 0-Day Attacks through Strategy-Based Detection Systems," *Security Privacy, IEEE*, vol. 11, no. 4, pp. 46–53, July 2013.

[32] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using EvoSuite," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, p. 8, 2014.

[33] M. Harman, "The Current State and Future of Search Based Software Engineering," in *2007 Future of Software Engineering*, ser. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 342–357. [Online]. Available: http://dx.doi.org/10.1109/FOSE.2007.29

[34] W. Afzal, R. Torkar, and R. Feldt, "A Systematic Review of Search-based Testing for Non-functional System Properties," *Information and Software Technology*, vol. 51, no. 6, pp. 957–976, Jun. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2008.12.005

[35] S. Türpe, "Search-Based Application Security Testing: Towards a Structured Search Space," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, March 2011, pp. 198–201.

[36] A. Avancini and M. Ceccato, "Security testing of web applications: A search-based approach for cross-site scripting vulnerabilities," in *Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on*, Sept 2011, pp. 85–94.

[37] J. Thomé, A. Gorla, and A. Zeller, "Search-based Security Testing of Web Applications," in *Proceedings of the 7th International Workshop on Search-Based Software Testing*, ser. SBST 2014. New York, NY, USA: ACM, 2014, pp. 5–14. [Online]. Available: http://doi.acm.org/10.1145/2593833.2593835

[38] C. Del Grosso, G. Antoniol, E. Merlo, and P. Galinier, "Detecting Buffer Overflow via Automatic Test Input Data Generation," *Computers & Operations Research*, vol. 35, no. 10, pp. 3125–3143, Oct. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.cor.2007.01.013

[39] S. Rawat and L. Mounier, "Offset-Aware Mutation Based Fuzzing for Buffer Overflow Vulnerabilities: Few Preliminary Results," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, March 2011, pp. 531–533.