Automatic Generation of Tests to Exploit XML Injection Vulnerabilities in Web Applications

Jan, Sadeeq; Panichella, Annibale; Arcuri, Andrea; Briand, Lionel

DOI
10.1109/TSE.2017.2778711

Publication date
2019

Document Version
Accepted author manuscript

Published in
IEEE Transactions on Software Engineering

Citation (APA)
Jan, S., Panichella, A., Arcuri, A., & Briand, L. (2019). Automatic Generation of Tests to Exploit XML Injection Vulnerabilities in Web Applications. *IEEE Transactions on Software Engineering*, *45*(4), 335-362. Article 8125155. https://doi.org/10.1109/TSE.2017.2778711

Automatic Generation of Tests to Exploit XML Injection Vulnerabilities in Web Applications

Sadeeq Jan*†, Annibale Panichella*, Andrea Arcuri*‡, Lionel Briand*
* Interdisciplinary Centre for Security, Reliability and Trust (SNT), University of Luxembourg
† University of Engineering & Technology Peshawar, Pakistan
‡ Westerdals Oslo ACT, Oslo, Norway
{jan, panichella, briand}@svv.lu, arcand@westerdals.no

Abstract—Modern enterprise systems can be composed of many web services (e.g., SOAP and RESTful). Users of such systems might not have direct access to those services, and rather interact with them through a single-entry point which provides a GUI (e.g., a web page or a mobile app). Although the interactions with such entry point might be secure, a hacker could trick such systems to send malicious inputs to those internal web services. A typical example is XML injection targeting SOAP communications. Previous work has shown that it is possible to automatically generate such kind of attacks using search-based techniques. In this paper, we improve upon previous results by providing more efficient techniques to generate such attacks. In particular, we investigate four different algorithms and two different fitness functions. A large empirical study, involving also two industrial systems, shows that our technique is effective at automatically generating XML injection attacks.

Index Terms—Evolutionary Testing; XML Injection; Security Testing

1 Introduction

The Service-Oriented Architecture (SOA) enables modular design [1]. It brings a great deal of flexibility to modern enterprise 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, especially when dealing with SOAP web services. 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. Although such vulnerabilities have been known for almost two decades, they are still very common in web applications and ranked first in the Open Web Application Security Project (OWASP) top ten, together with other code injection vulnerabilities[1]. This is due to many developers being not properly trained on security aspects [5] or being under intense time pressure to deliver software within a limited amount of time. It is therefore crucial to provide automated testing tools that support developers in triggering malicious attacks and detecting security vulnerabilities [5].

This paper focuses on the automated testing for XML injections (*XMLi*), a prominent family of attacks that aim at manipulating XML documents or messages to compromise XML-based applications. Our systems under test (SUTs) are the front-end web applications of SOA systems, as they are directly accessible from the internet and represent the main entry-points for *XMLi* attacks. With "front-end" we mean the main entry point on the server side of the SOA systems, and not code running in the browser (e.g., JavaScript frameworks for client-side HTML rendering like React and Angular). Among other functionalities, front-end web applications 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 [6], [7]). Such user inputs must be properly validated to prevent *XMLi* attacks. However, in the context of large web applications with hundreds of distinct input forms, some input fields are usually not properly validated [5]. Moreover, full data validation (i.e., rejection/removal of all potentially dangerous characters) is not possible in some cases, as meta-characters like "<" could be valid, and ad-hoc, potentially faulty solutions need to be implemented. For example, if a form is used to input the message of a user, emoticons such as "<3" representing a "heart" can be quite common. As a consequence, front-end web applications can produce malicious XML messages when targeted by *XMLi* attacks, thus compromising back-end services that consume these messages.

In practice, there exist approaches based on fuzz testing, e.g., ReadyAPI [8], WSFuzzer [9], 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 vulner-

1. https://www.owasp.org/index.php/Top_10_2013-A1-Injection

abilities that can be exploited only by using specific input strings in addition to the XML *meta-characters* [10], such as XML closing tags (e.g., `</test>`). 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 this paper, we propose an automated and scalable approach to search for test cases (attacks) that are effective at detecting *XMLi* vulnerabilities affecting the front-end web applications (our SUTs). To detect such vulnerabilities, we first identify a set of well-formed yet malicious XML messages that the SUTs can produce and send to back-end services. In the following, we refer to these malicious XML messages as *test objectives*, or TOs for brevity. The TOs are identified using SOLMI [10], a fully-automated tool that creates well-formed malicious XML messages based on known XML attacks and the XML schemas of the web services under test. Given a set of TOs, our testing goal is to verify whether such malicious XML messages can be generated by the SUT or whether the inputs validation and sanitization are able to prevent their generation. In addition, often there is no one-to-one mapping between inputs and outputs of the SUT since user inputs are typically processed and transformed before generating XML messages. Therefore, the goal is to search for user inputs that lead the front-end web application to generate each TO.

To solve this search problem, we use meta-heuristics to explore the input space of the SUT (e.g., text data in HTML input forms) in an attempt to generate XML messages matching the generated TOs. Search 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 targeted TOs. Our approach does not require access to the source code of the SUT and can, therefore, be applied in a black-box fashion on many different systems.

This paper is an extension of a previous conference paper [11], where we used standard genetic algorithms guided by the string edit distance (as fitness function) to exploit *XMLi* vulnerabilities. The contributions of this paper with respect to the conference paper are:

- We investigate four different search algorithms, namely Standard Genetic Algorithm (SGA), Real-coded Genetic Algorithm (RGA), Hill Climbing (HC) and Random Search (RS), while in the conference paper we compared only SGA and RS.
- We evaluate a different fitness function, namely the Real-coded Edit Distance (Rd), to overcome the limitations of the traditional String Edit Distance (Ed) in our context.
- We provide an in-depth analysis by comparing all possible combinations of fitness functions and search algorithms to determine the combination that is most effective and efficient in detecting *XMLi* vulnerabilities.
- We extensively analyze several co-factors that are likely to affect the effectiveness and efficiency of the proposed approach.

Based on the above contributions, we can then recommend to practitioners the best search algorithm and fitness function to uncover as many XML vulnerabilities as possible (effectiveness) while reducing the time needed to detect them (efficiency). Results are reported at a sufficient level of detail to enable other researchers to replicate our experiments and fully explain the results. We have carried out an extensive evaluation of the proposed approach by conducting two different case studies. In the first study, we compared all combinations of fitness functions and search algorithms with respect to the detection of *XMLi* vulnerabilities in (i) one open-source third-party application designed for secure-code training, and (ii) two web applications that interact with an industrial bank card processing system. We find that RGA combined with Rd is able to detect more *XMLi* vulnerabilities (better effectiveness) within a significantly lower amount of time (better efficiency) compared to the other combinations, including the one previously used in our conference paper [11], i.e., SGA with Ed.

To evaluate the applicability of our search-based approach in a realistic setting, we conducted a second case study involving two industrial systems. The first one is a web application having millions of registered users, with hundreds of thousands of visits per day. We focused on one of its pages with an HTML form. As our approach would be directly applicable to any system that receives HTTP messages, to show that this is indeed the case, our second case study involves a web service receiving JSON messages and generating XML messages for back-end SOAP services. Our results show that the proposed technique, when configured with RGA and Rd, successfully detects *XMLi* vulnerabilities in the evaluated industrial systems.

The remainder of the paper is structured as follows. Section 2 provides background information on *XMLi* attacks, and describes the testing context of our research. Section 3 describes our proposed approach and the tool that we developed for its evaluation. Section 4 reports and discusses our evaluation on two case studies including research questions, results and discussions. Further analyses regarding the various co-factors that may affect our results are presented in Section 6. Section 7 discusses related work. Threats to validity are discussed in Section 8. Finally, Section 9 concludes the paper.

2 Background

In this section, we briefly discuss *XMLi* attacks and describe our prior work [10] aimed at testing back-end web services.

2.1 XML Injection

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 attacks 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. We refer the reader to our previous work [10] for a more comprehensive categorization of *XMLi* attacks.

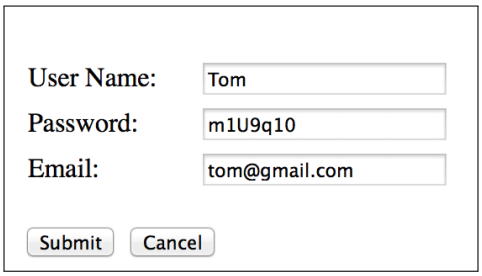


Fig. 1. The user registration web form having three input fields: User Name, Password, and Email.

Consider a simplified example in which users can register themselves through a web portal to a central service[2]. Once registered, a user can access different functionalities offered by the service. User registration data are stored in a database, accessed through a SOAP web service. Each user entry has a field 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 (shown in Figure 1) with three user input fields: *username*, *password*, and *email*. Each time a user submits a registration request, the application invokes the following piece of Java code to create an 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
10 validate(soapMessage);
```

Even though there is a validation procedure at Line 10, that 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 into the XML message in Figure 2. The *userid* element is replaced with a new element having the value "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. This message is well-formed and valid according to the associated XML schema (i.e., the XSD) and, therefore, the validation procedure does not help mitigating this vulnerability.

2. 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)

```
<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.

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" to launch an SQLi attack. If the application directly concatenates the received parameter values into an 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
```

2.2 Testing Web Applications for *XMLi* attacks

An SOA system typically consists of a front-end web application and a set of back-end web services. Messages are transmitted using the HTTP protocol while data are represented using the XML format [12]. The front-end application generates XML messages (e.g., toward SOAP and RESTful web services) upon incoming user inputs (as depicted in Figure 3). These XML messages are consumed by back-end systems or services, e.g., an SQL back-end, that are not directly accessible from the net.

Testing web applications for code injection attacks implies to thoroughly test both back-end services and front-end applications. Back-end services are typically protected by XML-gateways/firewalls, which are responsible for blocking incoming malicious requests. To provide an additional level of protection, front-end web applications contain *code-level defenses* [13], such as input validation and sanitization routines. In such a scenario, security analysts have to test the protection mechanisms both at back-end side (i.e., XML-gateways and firewalls) and front-end side (i.e., input sanitization and validation routines).

2.2.1 Testing the Back-end Web Services

In our previous work [10], we have developed a testing framework, namely SOLMI (SOLver and Mutation-based test generation for XML Injection), to automatically generate *XMLi* attacks able to bypass the XML gateways and target the back-end web services. SOLMI uses a set of mutation operators that can manipulate a non-malicious XML message to generate four types of *XMLi* attacks: *Type 1: Deforming*, *Type 2: Random closing tags*, *Type 3: Replicating* and *Type 4: Replacing*. The intent and impact of each of these *XMLi* attack types are different. *Type 1* attacks aim to create malformed XML messages to crash the system that processes 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 generate these types of attacks, SOLMI [10] relies on a constraint solver and attack grammars (e.g., an SQL grammar for nested SQL injection attacks). The constraint solver ensures that the malicious content, generated using the attack grammar, satisfies the associated constraints (e.g., Firewall rules) of the system. Therefore, the resulting XML messages are valid but malicious and are more effective in circumventing the protection mechanisms of the associated (back-end) web services.

In our prior work [10], we conducted an empirical study to assess the performance of SOLMI by testing 44 web services of a financial organization that are protected by an XML gateway (firewall). Our results showed that around 80% of the tests (i.e., XML messages with malicious content) generated by SOLMI were able to successfully bypass the gateway. Moreover, SOLMI outperformed a state-of-the-art tool based on fuzz testing, which did not manage to generate any malicious, bypassing *XMLi* attack [10].

2.2.2 Our Testing Context

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 (back-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.

Therefore, in this paper we focus on the front-end web application, which is our software under test (SUT). In particular, we aim to test whether the SUT 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 and PHP). Furthermore, this also helps broaden the applicability of our approach to systems in which source code is not 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.

3 Testing the Front-end Web Applications for *XMLi*: A Search-based Approach

If there exist inputs (e.g., data sent via forms from HTML pages) that can lead the SUT to generate malicious XML outputs, then the SUT is considered to be vulnerable. Thus, the malicious XML outputs generated by the SUT are our test objectives (TOs) and our goal is to search for user inputs generating them (see Figure 3). Consider the example of user registration given in Section 2.1: Figure 2 shows a possible TO where the *userid* element is manipulated, i.e., the original

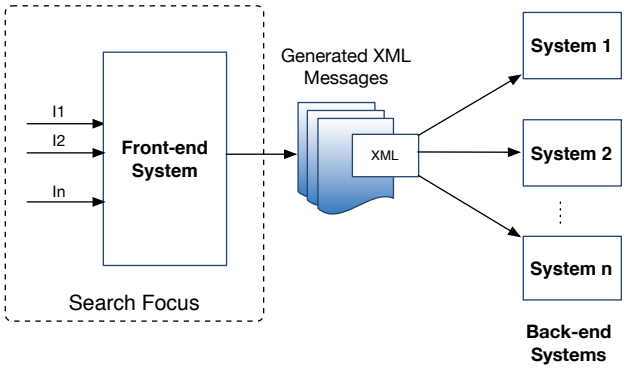


Fig. 3. Testing Context

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

A TO is said to be *covered* if we can provide inputs which result into the SUT producing the TO. Since the TO is malicious by design, the SUT is not expected to do so unless it is vulnerable. In other words, we search for user inputs that can lead the SUT to generate malicious messages and send them to the web services behind the corporate firewall/gateway, which are not directly accessible to an attacker. Sending such TOs to the backend systems/services could severely impact them depending on the malicious content that these TOs carry.

More generally, the problem of testing the front-end web application for *XMLi* attacks can be defined as follows:

***Problem* 3.1:** *Let $TO = \{TO_1, \ldots, TO_k\}$ be the set of test objectives for a given SUT. Our problem is to find a set of test inputs $T = \{T_1, \ldots, T_k\}$ such that SUT(T_i)=TO_i, for each $TO_i \in TO$.*

In the definition above, SUT(T_i) denotes the XML message produced by the SUT when executed with T_i.

Although the TOs are known a priori, in practice we do not know whether they can be actually generated by the SUT, i.e., whether they are feasible or not. Since we consider the SUT as a black-box, we also do not know a priori how inputs are related to output XML messages. It is also worth noticing that there is likely no one-to-one mapping between user inputs and XML outputs, as inputs are processed and transformed by input sanitization and validation routines of the front-end web applications.

3.1 Test Objectives Generation

In our approach, TOs are XML messages satisfying the following conditions: (i) they are syntactically valid; (ii) they are conforming to their given schema (e.g., in XSD format); (iii) they contain malicious content. The first two properties are necessary, otherwise wrongly-formatted XML messages would be trivially discarded by the XML-gateway/firewall protecting the web services. Furthermore, the set of TOs should be diverse such as to cover the four types of *XMLi* attacks described in Section 2.2.1.

For these reasons, in this paper we use the tool SOLMI [10] to produce the TOs. In addition to satisfying all the aforementioned properties, we would like that, if they

are generated by the SUT, they also have high chances of bypassing the XML gateway/firewall. SOLMI also ensures diversity in the generated TO set, which is important for the success of our technique. Having a diverse set of TOs increases our chances of finding TOs that can be produced by the SUT. Therefore, when generating TOs, we make sure that each XML element and attribute of the messages is modified at least once with all the types of *XMLi* attacks, when possible, as described in Section 2.2.1.

3.2 Search-Based Testing

In our context, applying search-based techniques requires us to address three issues [14]: (i) choose an encoding schema to represent candidate solutions (i.e., test inputs); (ii) design a fitness function to guide the search for malicious test inputs; and (iii) choose and apply an effective search algorithm to generate inputs closer to the target TO. Our choices for the aforementioned tasks are detailed in the next sub-sections.

3.2.1 Solution Encoding

A given SUT requires N input parameters to be submitted for producing XML messages that will be sent through HTTP to the web services. Therefore, the search space is represented by all possible tuples of N strings that can be submitted via the web form, with one string for each single parameter. In this context, a string is a sequence of alphanumeric and special characters (e.g., %, || or &) that can be inserted by an attacker in the web form.

Therefore, we use the following encoding schema: a candidate test case for the SUT with N input parameters is a tuple of strings $T = \langle S_1, S_2, \ldots, S_N \rangle$ where a S_i denotes the string for the i-th input parameter of the SUT. A generic string in T is an array of k characters, i.e., $S_i = \langle c_1, c_2, \ldots, c_k \rangle$. The length k of the array is fixed based on the expected maximum length of the corresponding input parameter. To allow input strings with different length, we use a special symbol to denote the "empty" character, i.e., absence of character. In this way, the lengths of input strings can vary during the search even if the length of the array (i.e., k) in the encoding schema is fixed. In other words, the array $S_i = \langle c_1, c_2, \ldots, c_k \rangle$ can be filled with the "empty" character to represent shorter strings.

Theoretically, characters in the input string can come from the extended ASCII code as well as from UNICODE. However, in this paper we consider only printable ASCII characters with code between 32 and 127 since, as noticed by Alshraideh et al. [15], the majority of software programs do not use characters outside this range (i.e., non-printable characters).

3.2.2 Fitness Function

The effectiveness of the search strongly depends on the guidance of the fitness function, which evaluates each candidate solution T according to its closeness to the target TO. In particular, when a candidate solution T is executed against the SUT, it should lead to the generation of an XML message that matches the TO. Hence, the fitness function is the distance $d\,(\text{TO}, \text{SUT(T)})$ between the target TO and the XML message that the SUT produces upon the execution of T, i.e., SUT(T). The function $d(\cdot)$ can be any distance measure such that $d\,(\text{TO}, \text{SUT(T)}) = 0$ if and only if SUT(T) and the TO are identical, otherwise $d\,(\text{TO}, \text{SUT(T)}) > 0$. In this paper, we investigate two different measures for the fitness function: the *string edit distance* and the *real-coded edit distance*.

String Edit Distance. The first fitness function is the Levenshtein distance, which is the most common distance measure for string matching. Its main advantage compared to other traditional distances for strings (e.g., Hamming distance) is that it can be applied to compare strings with different lengths [16]. In our context, the length of the XML messages generated by the SUT varies depending on the input strings (i.e., the candidate solution T) and the data validation mechanisms in place to prevent possible attacks. Therefore, the edit distance is well suited for our search problem. In addition, it has been shown in the literature [15] that this distance outperforms other distance measures (e.g., Hamming distance) in the context of test case generation for programs with string input parameters, despite its higher computational cost[3].

In short, the Levenshtein distance is defined as the minimum number of editing operations (inserting, deleting, or substituting a character) required to transform one string into another. More formally, let A_n and B_m be two strings to compare, whose lengths are n and m, respectively; the edit distance is defined by the following recurrence relations:

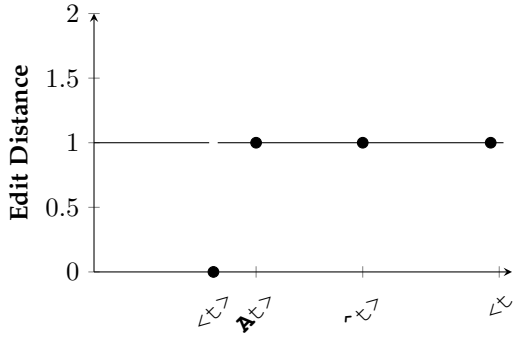
$$d_E(A_n, B_m) = \min \begin{cases} d_E(A_{n-1}, B_m) + 1 \\ d_E(A_n, B_{m-1}) + 1 \\ d_E(A_{n-1}, B_{m-1}) + f(a_n, b_m) \end{cases} \quad (1)$$

where a_n is the n-th character in A_n, b_m is the m-th character in B_m, and $f(a_n, b_m)$ is zero if $a_n = b_m$ and one if $a_n \neq b_m$. In other words, the overall distance is incremented by one for each character that has to be added, removed or changed in A_n to match the string B_m. The edit distance takes values in $[0; \max\{n, m\}]$, with minimum value $d_E = 0$ when $A_n = B_m$ and maximum value of $d_E = \max\{n, m\}$ when A_n and B_m have no character in common.

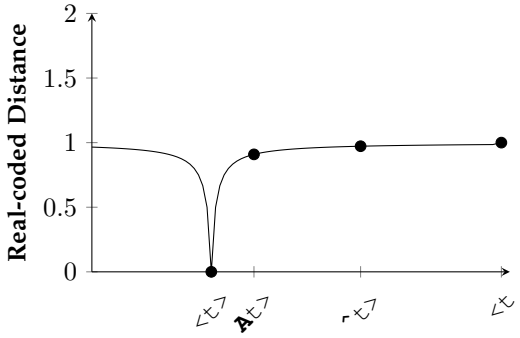
To clarify, let us consider the following example of a TO and a SUT with one single input parameter. Let us assume that the target TO is the string `<test>data OR 1=1</test>`; and let us suppose that upon the execution of the test $T = \langle$`OR %`$\rangle$, the SUT generates the following XML message SUT(T) = `<test>data OR %</test>`. In this example, the edit distance $d_E\,(\text{TO}, \text{SUT(T)})$ is equal to three, as we need to modify the "`%`" character into "`1`", and then add the two characters "`=1`" for an exact match with the TO.

One well-known problem of the edit distance is that it may provide little guidance to search algorithms because the fitness landscape around the target string is largely flat [15]. For example, let us consider the target TO = `<t>` and let us assume we want to evaluate the three candidate tests T_1, T_2 and T_3 that lead to the following XML messages: SUT(T_1) = `At>`, SUT(T_2) = `^t>`, SUT(T_3) = `<t`. The messages SUT(T_1) and SUT(T_2) share two characters with the

3. The computational cost of the edit distance is $O(n \times m)$, where n and m are the lengths of the two strings being compared.



(a) Edit distance



(b) Real-coded edit distance

Fig. 4. Fitness landscapes for the *edit distance* and the *real-coded edit distance* for the target string TO = <t>

target TO (i.e., t and >) and have a correct length, i.e., three characters. Instead, the message SUT(T_3) shares two characters with the target TO but it is one character shorter. Therefore, we may consider T_1 and T_2 to be closer to the target TO than T_3 since they have the correct number of characters, two of which match the TO. However, using the edit distance, all the three tests will have the same distance to the TO since they require to change only one character, i.e., $d_E(\texttt{<t>}, \texttt{\textbf{A}t>}) = d_E(\texttt{<t>}, \texttt{\^{}t>}) = d_E(\texttt{<t>}, \texttt{<t}) = 1$. In this example, the edit distance is not able to distinguish between messages having the correct length (e.g., T_1) and messages that are shorter or longer than the TO (e.g., T_3).

In general, the fitness landscape around the target TO will be flat as depicted in Figure 4-(a): all strings that require changing (e.g., **A**t>), add (e.g., <t) or remove (e.g., <t**t**>) one character will have a distance d_E equal to 1 while the distance will be 0 for only one single point. Thus, a search algorithm would have to explore this whole, very large neighborhood, without any particular guidance.

Real-Coded Edit Distance. It is a variant of the Levenshtein distance that we introduce to overcome the limitations of the original distance, as described in the previous paragraphs. Given a target TO, the set of strings at a given distance D is extremely large. For example, let us consider a TO with n characters selected from an alphabet Ω; there are $2 \times n \times |\Omega|$ strings having a Levenshtein distance D=1: (i) n strings that differ for one single missing character; (ii) $n \times |\Omega|$ strings containing one additional (spare) character; (iii) $n \times (|\Omega| - 1))$ strings that require to replace one single character to perfectly match the TO. Although we restrict the alphabet to the printable ASCII characters with code between 32 and 127, the number of strings with a Levenshtein distance D=1 for a target TO with 100 characters is $100 \times 2 \times (127 - 32 + 1)) = 19200$. In such a large neighborhood, the Levenshtein distance does not provide effective guidance to the search algorithms.

Therefore, we modify the Levenshtein distance to focus the search on sub-regions of the large neighborhood of the target TO. In particular, we consider the relative distance between characters in the ASCII code, which helps replacing the plateaus with a gradient. This is done by changing the recurrence relations as follows:

$$d_R(A_n, B_m) = \min \begin{cases} d_R(A_{n-1}, B_m) + 1 \\ d_R(A_n, B_{m-1}) + 1 \\ d_R(A_{n-1}, B_{m-1}) + \frac{|a_n - b_m|}{1+|a_n - b_m|} \end{cases} \quad (2)$$

In Equation 2, the first two recurrence rules are identical to the traditional edit distance covering the case where A_n will match B_m by removing or adding one character, respectively. The change is applied on the third recurrence rule, which covers the case when the character a_n should be replaced by the character b_m. In the traditional edit distance, if a_n is not equal to b_m then the overall distance is always incremented by one. Instead, in Equation 2, if a_n is not equal to b_m, then the overall distance is incremented by the factor $\mid a_n - b_m \mid$, which is the absolute value of the difference between the ASCII codes for the two characters a_n and b_m. Such an increment factor is normalized using the well-known normalization function $\phi(x) = x/(x+1)$ to obtain distance values within the interval $[0; 1]$.

To describe the benefits of this new distance, let us consider the example used previously to describe the fitness landscape of the edit distance: the target TO is the string <t>, and the tests to evaluate lead to the XML messages SUT(T_1) = At>, SUT(T_2) = ^t>, SUT(T_3) = <t. Using the real-coded edit distance, we obtain:

$$d_R(\texttt{<t>}, \texttt{\textbf{A}t>}) = \frac{\mid 60 - 65 \mid}{\mid 60 - 65 \mid +1} \approx 0.8333$$
$$d_R(\texttt{<t>}, \texttt{\^{}t>}) = \frac{\mid 60 - 94 \mid}{\mid 60 - 94 \mid +1} \approx 0.9714$$
$$d_R(\texttt{<t>}, \texttt{<t}) = 1$$

Thus, with d_R the three tests are not equally distant to the TO anymore: the test T_3 is the furthest one among the three tests. Therefore, differently from the edit distance, the new distance is able to distinguish between messages with correct length (e.g., T_1) and messages that are longer or shorter than the TO (e.g., T_3). We can also observe that the new distance returns two different values for T_1 and T_2 although both the two tests need to replace only one character to perfectly match the TO. This difference is given by the relative distance between the character to match in the TO (i.e., "<") and the two characters to replace (i.e., "A" for T_1 and "^" for T_2) according to their ASCII codes. Therefore, d_R introduces a distance among characters such that small differences among strings can still lead to differences in fitness values.

Figure 4-(b) plots the fitness landscape around the target TO=<t> as well as the three candidate tests T_1, T_2, and T_3 in the previous example. In particular, the x axis orders the candidate XML messages according to their ASCII codes

while the y axis reports the corresponding fitness function values produced by the real-coded edit distance. As we can observe, the plateaus are replaced by a fitness function providing more guidance by considering the ordering of the characters according to their ASCII codes. Though such a distance may seem arbitrary, it helps the search focus on sub-regions of the neighborhood of the target TO, e.g., by preferring T_1 over T_2 in the example.

In theory, the usage of the ASCII code is not mandatory and we could use any arbitrary mapping $f : c \rightarrow \mathbb{N}$ between characters (c) and integers ($\mathbb{N}$). Given a mapping f, the relative distance between two characters a_n and b_n can be always computed using the formula $\phi(|f(a_n) - f(b_n)|)$. Using any other mapping f would lead to the same fitness landscape transformation: the plateaus of the Levenshtein distance are replaced by a gradient, thus providing a "direction" to follow to match the target TO. Among all possible mappings, in this paper we opted for the ASCII code because it is a well-known standard encoding. Moreover, we did not observe any difference in our empirical results when replacing the ASCII code with some other character mappings.

As for any heuristic, the new distance might have some side effects (e.g., create new local optima). Therefore, such a distance needs to be empirically evaluated to check if indeed it provides the benefits we expect from the theory.

3.2.3 Solvers

Once the encoding schema and the fitness function are defined, search algorithms can be applied to find the optimal solutions, as for example malicious input strings in our case. Various search algorithms have been proposed in the literature to solve different software engineering problems [14]. However, there does not exist a search algorithm that outperforms all other algorithms for all possible optimization problems [17]. According to the *no free lunch theorem* [17], if an algorithm A outperforms another algorithm B for a problem P_1, there exists another problem P_2 for which B outperforms A. Therefore, we need to investigate different search algorithms to better understand which one works better for our problem, namely generating *XMLi* attacks.

In this paper, we investigate four different search algorithms, which are (i) random search (RS), (ii) hill climbing (HC), (iii) the standard genetic algorithm (SGA), and (iv) the real-coded genetic algorithm (RGA). These algorithms are designed to solve different types of search problems, such as functions with plateaus, unimodal functions (i.e., with only one optimum), multimodal functions (i.e., with multiple local optima), and problems whose solution encoding contains numbers (real-coded). In the following paragraphs, we describe each search algorithm as well as the type of problems for which it outperforms the other algorithms.

Random Search (RS) is the simplest search algorithm, which uniformly samples the search space by evaluating random points. It starts with a randomly generated test T representing the initial candidate solution to the problem. Such a solution is evaluated through the fitness function and it is stored until a new, better solution is found in the next iterations of the algorithm, or if a stop condition is reached. At each iteration, a new test T^* is randomly generated and compared with T using the fitness function. If T^* has a better fitness value than T, then it is kept as current solution for the next iterations, i.e., $T = T^*$. The search ends after a fixed number of iterations. The final test T will be the best solution among all those observed across all the iterations.

In our context, a randomly generated solution $T = \langle S_1, S_2, \ldots, S_N \rangle$ is composed of N arrays of characters with a fixed length k. Each array S_i contains characters randomly taken from the set of available characters (alphabet), including the "empty" character. Therefore, a solution T is composed of arrays representing strings with variable lengths $\leq k$. Notice that the solutions generated by RS are inputs for web forms and not complete XML messages (TOs). Indeed, the XML messages are generated by the SUT, which fills predefined XML message templates with the input strings processed by the sanitization and validation routines. Therefore, although the inputs are randomly generated, the corresponding XML messages are syntactically correct as they are generated by the SUT, which creates well-formed XML structure.

Since RS does not refine previously generated solutions, it has usually a low probability to reach the global optimum. However, it is often used in the software engineering literature as baseline for comparison with more advanced algorithms. Moreover, RS is very effective to optimize problems whose fitness landscape contains plateaus (e.g., in the case of the edit distance) and provides poor guidance [18]. For example, it has been shown to outperform other search algorithms (e.g., evolutionary algorithms) when solving specific problems, such as automated software repair [19], and hyper-parameter optimization [20].

The Simple Hill Climbing (HC) is a local search algorithm, which iteratively exploits the *neighborhood* of the current solution to find better nearby solutions (*neighbors*). Similar to random search, hill climbing starts with a single randomly generated test T, which represents the current solution to the problem. At each iteration, a new solution T^* is taken from the neighborhood of T and evaluated against the fitness function. If T^* improves the fitness function, then it becomes the current solution for the next iteration, i.e., $T = T^*$. The search ends after a fixed number of iterations or if a zero-fitness value is reached, which indicates that the target TO is matched.

The key ingredient for the HC algorithm is the definition of the *neighborhood*, which corresponds to the set of tests (neighbors) that can be obtained from the current solution T by applying "small mutations". Let $T = \langle S_1, S_2, \ldots, S_N \rangle$ be the current test composed of N arrays of characters. A neighbor is obtained from T by mutating its constituent arrays using one of the following operators: *adding*, *replacing* or *deleting* characters. Each operator is performed with probability $p = 1/3$, i.e., the three operators are mutually exclusive (only one operator is applied at a time) and equiprobable. Given an array of characters $S_i = \langle c_1, c_2, \ldots, c_k \rangle$ of length k, the three operators are implemented as follows:

- *deleting*: a character c_j in S_i is deleted with probability $p_d = 1/k$. The deletion is performed by replacing the character $c_j \in S_i$ with the "empty" character, which is then shifted to the end of the array. As explained in Section 3.2.1, the "empty" character represents the

absence of character and is used to allow input strings with different length during the search.

- *replacing*: a character c_j in S_i is replaced with a new character c_j^* with probability $p_r = 1/k$, where c_j^* is randomly selected from the set of printable ASCII characters.
- *adding*: a new character c^* is inserted in S_i at a random position $j \in [1, \ldots, k]$, the subsequent characters at that position are shifted to the right, and the "empty" character at the last position in the array is deleted to accommodate the newly inserted character. The new character is added if and only if there exists at least one "empty" character in the array. This restriction ensures that inserting a new character does not delete an existing "non-empty" character as it is the case with the "replacing" operator.

Therefore, on average only one character is removed, replaced or added in the arrays S_i contained in the test T.

Despite its simplicity, HC is very effective when the fitness forms an unimodal function in the search space, i.e., functions with only one single optimal point [21]. If the fitness function provides good guidance, HC converges faster to the optimal point compared to global search algorithms (e.g., evolutionary algorithms) and RS. However, it can return sub-optimal solutions for multimodal functions since it converges to the first local optimum encountered during the search, even if it is not the global one [21].

In this paper, we consider the *simple* HC algorithm over the other variants that have been proposed in the literature [21]. One of its most efficient variants is the *steepest descent hill climbing* [21]. Given a trial solution T, this variant examines all possible *neighbors* of T and selects as new solution the deepest descend, i.e., the neighbor with largest fitness function improvement. In our context, we could not use this variant since it requires to analyze the entire neighborhood of T, whose size is significantly large. Indeed, the number of strings that can be obtained by adding, replacing or deleting one single character from T is $2 \times n \times |\Omega|$, where n is the length of T and $|\Omega|$ is the size of the alphabet (see Section 3.2.2). For this reason, we opted for the simple variant of the HC, which is easier to implement, yet efficient as it fast converges toward optimal solutions [21].

Standard Genetic Algorithm (SGA) is a metaheuristic solver inspired by the mechanisms of natural selection and adaptation. In a nutshell, it starts with a pool of solutions, called *population*, where each solution (or *chromosome*) is a randomly generated test. Then, the population is iteratively evolved by applying well-known genetic operators, namely *crossover*, *mutation* and *selection*. At each iteration (*generation*), pairs of solutions (*parents*) are selected and recombined using the *crossover* operator, which creates new solutions (*offspring*) to form the population for the next generation. Other than inheriting parts (*genes*) from their parents, offspring are further modified, with a given small probability, using the *mutation* operator. Solutions are selected according to a *selection* operator, which typically gives higher selection probability to solutions in the current population with higher fitness values (fittest individuals). This process is repeated until a zero-fitness value is achieved (i.e., the TO is matched) or after a fixed number of generations.

The most "popular" (i.e., widely used in the literature, because for example they give good results on average and/or are easier to implement) genetic operators in SGA are the *binary tournament selection*, the *multi-point crossover* and the *uniform mutation* [21]. They are defined as follows:

- the *binary tournament selection* is the most common selection mechanism for GAs because of its simplicity and efficiency [22], [23]. With this operator, two individuals are randomly taken from the current population and compared against each other using the fitness function. The solution with the best fitness value wins the tournament and is selected for reproduction.
- the *multi-point crossover* generates two offspring O_1 and O_2 from two parent solutions P_1 and P_2 by recombining their corresponding arrays of characters. More precisely, let $P_1 = \langle S_1, S_2, \ldots, S_N \rangle$ and $P_2 = \langle R_1, R_2, \ldots, R_N \rangle$ be the two selected parents, the two offspring O_1 and O_2 are generated as follows:

$$O_1 = \langle \otimes(S_1, R_1, p_1), \ldots, \otimes(S_N, R_N, p_N) \rangle \quad (3)$$

$$O_2 = \langle \otimes(R_1, S_1, p_1), \ldots, \otimes(R_N, S_N, p_N) \rangle \quad (4)$$

  where the generic element $\otimes(S_i, R_i, p_i)$ denotes the array obtained by cutting the two arrays S_i and R_i at the same random cut point p_i and then concatenating the head part from S_i with the tail part from R_i. Similarly, $\otimes(R_i, S_i, p_i)$ indicates the array obtained by applying the same random cut point p_i but concatenating the head part from R_i with the tail part from S_i. Therefore, the i-th array from one parent is recombined with the corresponding array at the same position i in the other parent.
- the *uniform mutation* is finally used to mutate, with a small probability, newly generated solutions in order to preserve diversity [21]. It corresponds to the mutation operator used for the hill climbing algorithm when generating neighbors: tests are mutated by deleting, replacing or adding characters in the corresponding array of characters.

GAs are global search algorithms and are thus more effective than local search solvers for multimodal problems. This is because they use multiple solutions to sample the search space instead of a single solution (e.g., for the hill climbing) which could bias the search process [21]. On the other hand, GAs can suffer from a slower convergence to the local optimum when compared to hill climbing. Therefore, they are usually less effective and efficient for unimodal problems [21].

Real-Coded Genetic Algorithm (RGA) is a variant of GAs designed to solve numerical problems with real or integer numbers as decision variables (genes) [24]. The main difference between SGA and RGA is captured by the genetic operators that are used to form new solutions. In SGAs, the crossover creates offspring by exchanging characters from the parents and, as a result, the new solutions will only contain characters that appear in the parent chromosome. Further, in SGA, diversity is maintained by the mutation operator, which is responsible for replacing characters inherited from the parents with any other character in the alphabet. Instead, in RGA, the parents are recombined by applying numerical functions (e.g., the arithmetic mean) to

create offsprings that will contain new numbers (i.e., genes) not appearing in the parent chromosomes. Mutation, on the other hand, alters solutions according to some numerical distribution, such as a Gaussian distribution.

In this paper, we investigate the usage of RGAs since they have been shown to be more effective than SGAs when solving numerical and high dimensional problems [24], [25]. In particular, our problem is numerical if we consider characters as numbers in ASCII code (as in the real-coded edit distance) and it is high dimensional (the number of dimensions corresponds to the length of the chromosomes). Indeed, maintaining the same encoding schema used for SGA, each array of characters $S_i = \langle c_1, c_2, \ldots, c_k \rangle$ of a test T can be converted in an array of integers $U_i = \langle u_1, u_2, \ldots, u_k \rangle$ such that each $u_i \in U$ is the ASCII code of the character $c_i \in S$ when applying real-coded crossover or mutation.

Popular genetic operators for RGA are the *binary tournament selection*, the *single arithmetic crossover* [26], and *Gaussian mutation* [27]. Therefore, the selection mechanism is the same as in SGA, whereas crossover and mutation operators are different. Before applying these two numerical operators, we convert the input strings forming a test T in arrays of integers by replacing each character with the corresponding ASCII code. Once new solutions are generated using the *single arithmetic crossover* and *gaussian mutation*, the integer values are reconverted into characters.

The *single arithmetic crossover* is generally defined for numerical arrays with a fixed length. For example, let $A = \langle a_1, a_2, \ldots, a_k \rangle$ and $B = \langle b_1, b_2, \ldots, b_k \rangle$ be two arrays of integers to recombine; it creates two new arrays A' and B' as copies of the two parents and modify only one element at a given random position i using the arithmetic mean. In other words, A' and B' are created as follows [26]:

$$A' = \langle a_1, a_2, \ldots, a'_i, \ldots a_k \rangle \quad (5)$$

$$B' = \langle b_1, b_2, \ldots, b'_i, \ldots b_k \rangle \quad (6)$$

where the integers a'_i and b'_i are the results of the weighted arithmetic mean between $a_i \in A$ and $b_i \in B$; and $i \leq k$ is a randomly generated point. The weighted arithmetic mean is computed using the following formulae [26]:

$$a'_i = a_i \cdot \rho + b_i \cdot (1 - \rho) \quad (7)$$

$$b'_i = b_i \cdot \rho + a_i \cdot (1 - \rho) \quad (8)$$

where ρ is a random number $\in [0; 1]$. Finally, the two resulting real numbers a'_i and b'_i are rounded to their nearest integers.

In our case, parent chromosomes are tuples of strings and not simple arrays of integers. Therefore, we apply the single arithmetic crossover for each pair of arrays composing the two parents, after the conversion of the characters to their ASCII codes. More formally, let $P_1 = \langle S_1, S_2, \ldots, S_N \rangle$ and $P_2 = \langle R_1, R_2, \ldots, R_N \rangle$ be the two selected parents; the two offsprings O_1 and O_2 are generated as follows:

$$O_1 = \langle \mu(S_1, R_1, p_1), \ldots, \mu(S_N, R_N, p_N) \rangle \quad (9)$$

$$O_2 = \langle \mu(R_1, S_1, p_1), \ldots, \mu(R_N, S_N, p_N) \rangle \quad (10)$$

where S_i is the array of ASCII codes in position i from the parent P_1; R_i is the array of ASCII codes in position i from the parent P_2; the elements $\mu(S_i, R_i, p_i)$ and $\mu(R_i, S_i, p_i)$ are the two arrays created by the single arithmetic crossover on S_i and R_i with random point $p_i \in [0; 1]$.

The *gaussian mutation* is similar to the uniform mutation for SGA. Indeed, each test T in the new population is mutated by deleting, replacing or adding characters in the corresponding array of characters. The main difference is represented by the routine used to replace each character with another one. With the uniform mutation, a character is replaced with any other character in the alphabet. Instead, the gaussian mutation is defined for numerical values, which are replaced with other numerical values but according to a Gaussian distribution [27]. In our case, let $S_i = \langle c_1, c_2, \ldots, c_k \rangle$ be the array of ASCII codes to mutate; each ASCII code c_j in S_i is replaced with a new ASCII code c_j^* with probability $p_r = 1/k$. The integer c_j^* is randomly generated using the formula:

$$c_j^* = c_j + c_j \cdot \delta(\mu, \sigma) \quad (11)$$

where $\delta(\mu, \sigma)$ is a normally distributed random number with mean $\mu = 0$ and variance σ [27]. In other words, the new ASCII code is generated by adding a normally distributed delta to the original ASCII code c_j. The remaining issues to solve include (1) this mutation scheme generates real numbers and not integers and (2) the generated numbers can fall outside the range of printable ASCII code (i.e., outside the interval $[32; 127]$). Therefore, we first round c_j^* to the nearest integer number. Finally, the mutation is cancelled if the new character c_j^* is lower than 32 or greater than 127.

4 Empirical Studies

This section describes our empirical evaluation whose objective is to assess the proposed search-based approach and compare its variants in terms of different fitness functions and search algorithms, as discussed in Section 3.

4.1 Study Context

The evaluation is carried out on several front-end web applications grouped into two case studies. The first study is performed on small/medium web applications, whereas the second one involves subsets (e.g., processing a specific HTML page) of industrial systems. These two studies are described in detail below.

Study 1. The first case study involves three subjects with various web-applications. The first two subjects are SBANK and SecureSBANK (SSBANK), which contain web applications interacting with a real-world bank card processing system. They are simplified versions of the actual front-end web applications from one of our industrial collaborators (a credit card processing company[4]). The SBANK and SSBANK versions used in our study contain the HTML forms and input processing routines from the original web applications as they represent the actual code under test. However, we disabled other routines, such as logging, encryption and other routines responsible for the interactions with the back-end web services. Finally, the back-end services were replaced by mock-up services to not compromise them during our testing process.

4. The name of the company cannot be revealed due to a non-disclosure agreement

```
<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*.

Both SBANK and SSBANK have three applications that differ regarding their number of user inputs, ranging from one to three user inputs. Note that all TOs in our empirical study involve up to three user inputs. Hence, the additional TOs that SOLMI could generate for forms with more than three inputs would just be similar malicious strings as for forms with three inputs but applied to different subsets of inputs. In the following, we will refer to SBANK1 (or SSBANK1), SBANK2 (or SSBANK2), and SBANK3 (or SSBANK3) for the applications with one, two, and three user inputs, respectively. These different versions of the same applications are used to analyze to what extent the number of input parameters affects the ability of solvers and fitness functions to detect XMLi vulnerabilities (see Section 6 for further details).

Each SBANK/SSBANK application receives user inputs, produces XML messages and sends them to the web services of the card processing system. An example of XML message produced by an SBANK/SSBANK application is depicted in Figure 5. Such a message contains four XML elements, which are *UserName*, *IssuerBankCode*, *CardNumber*, and *RequestID*. The first three elements are formed using the submitted user inputs while the *RequestID* element is generated by the application automatically. In other words, the application logic does not allow users to tamper with the value of this element unless they do so maliciously.

Applications in SBANK are vulnerable to XML injections as there is no validation or sanitization of the user inputs. The SSBANK applications are similar to SBANK except that one of the input parameters is validated, i.e., the application checks the input data for malicious content. Before producing the XML message, the latter applications validate the user input parameter *IssuerBankCode* and generate an error message if any malicious content is found. These two applications allow us to assess, in a controlled fashion, the impact of input validation procedures on the ability of solvers and fitness functions to detect *XMLi* vulnerabilities.

The third subject of our first study is *XMLMao*, an open source web application that is deliberately made vulnerable for testing XML injection attacks [28]. It is part of the *Magical Code Injection Rainbow (MCIR)* [28] framework for building a configurable vulnerability test-bed. This application accepts a single user input and creates XML messages. It has 1178 lines of code written in PHP. We chose to include such an open source application in our evaluation to have, as part of our study, a publicly accessible system that future research can use as a benchmark for comparison.

Study 2. The second study consists of two subjects provided by one of our industrial collaborators which, to preserve confidentiality, are referred to by arbitrary names: M and R. 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. R is one of the RESTful web services interacting with M. This web service receives requests in JSON format, and interacts with two SOAP web services and one database. R is significantly smaller than M, as it consists only of around five thousand lines of code. Of the different API methods provided by R, in this paper we focus on a POST that takes as body a JSON object with three string fields.

As the experiments on these two systems had to be run on a dedicated machine (e.g., they could not be run on a research cluster of computers) due to confidentiality constraints, we could not use all of theirs web pages and endpoints. We chose those two examples manually, by searching for non-trivial cases (e.g., web pages with at least two string input parameters that are not enumerations), albeit not too difficult to analyze, i.e., given the right inputs, it should interact with at least one SOAP web service. Due to non-disclosure agreements and security concerns, no additional details can be provided on M and R.

In this work, we analyze security testing from the point of view of users sending malicious data inside HTML forms. Sending data from the browser to the SUT is done on a TCP connection, using the HTTP protocol. In particular, this is done with a POST method with the data of the input form as HTTP body payload, encoded in the *x-www-form-urlencoded* format. However, an attacker does not need to use the browser: s/he can just directly open a TCP connection to the SUT, and craft a valid HTTP POST message. And this is what we do with our tool, where we use search to generate the right string data for the variables of the HTTP payloads. Therefore, our technique can be used for any SUT that takes HTTP messages as input. Besides web applications, web services also fit such a description. R is a RESTful web service, where inputs come from HTTP messages. So, we can use our technique directly on it. We just need to handle the JSON format instead of *x-www-form-urlencoded* when creating the payload of the messages we send. The use of R in our empirical study is mainly to show that our technique can be used also in other contexts besides HTML forms.

The main difference between Study 1 and Study 2 is the complexity of the code invoked by web pages rather than the size of their forms. Even if we test single pages from a large, industrial system, there is still going to be substantial functionality and code executed, contributing to the time consumed by the testing process. The systems selected in Study 2 led to lengthy experiments, as we could not use an HPC platform for an industrial system.

Test Objectives (TOs). For each application and for each case study, we created the target TOs based on the four types of XML injection attacks described in Section 3.1. Table 1 reports on the number of generated TOs collected

TABLE 1
Description of Test Objectives

App. Name	#Input	#TOs per Attack				Total #TOs
		Type 1	Type 2	Type 3	Type 4	
SBANK	1	3	3	3	0	9
	2	3	3	3	1	10
	3	3	3	3	1	10
SSBANK	1	3	3	3	0	9
	2	3	3	3	1	10
	3	3	3	3	1	10
XMLMao	1	4	4	4	0	12
M	2	1	1	1	1	4
R	3	1	1	1	1	4

per study subject and type of XML injection attacks. For SBANK/SSBANK applications with two and three input parameters, there are ten TOs in total: three TOs (one for each attack of types *Type 1-Type 3*) for each of the three XML elements and one additional TO for the *Type 4* attacks. Note that the *Type 4: Replacing* attack is a more advanced form of XML injection that requires at least two XML elements where the value of one of them must be auto-generated by the application. Therefore, this attack can be applied only to the *RequestID* element as it is the only auto-generated element in the application. Moreover, this attack should not be applied to web applications with only one input parameter, otherwise the resulting TO will be unfeasible. As a consequence, for SBANK/SSBANK applications with only one input parameter, we have nine TOs in total, corresponding to the attacks of types *Type 1-Type 3* for each of the three XML elements.

The XMLMao application has one user input that can be inserted in four possible locations in the generated XML messages. Therefore, we create TOs by applying each type of attack on the four XML elements. As depicted in Table 1, we do not have TOs for the attack of *Type 4: Replacing* because XMLMao has only one input parameter. In total, we obtain 12 TOs (3 attacks $\times$ 4 locations) for this subject. Finally, for the industrial applications (M and R), we have four TOs, i.e., one TO for each type of attack.

4.2 Research Questions

Our evaluation addresses the following research questions:

- ***RQ1****: What is the best fitness function for detecting XMLi vulnerabilities?* With this first research question, we aim at comparing the two fitness functions defined for the XMLi vulnerability detection problem. In particular, we compare the performance of each solver (e.g., hill climbing), considered individually, when used to optimize the *real-coded edit distance* proposed in this paper and the traditional *string edit distance* [15], [11]. In particular, the comparison is performed in terms of the number of generated TOs via the SUT (effectiveness) and the time needed to generate them (efficiency). In our context, the generation of a TO via the SUT implies the detection of an *XMLi* vulnerability. Therefore, in answering this research question, we consider the following two sub-questions:

  RQ1.1 *[Effectiveness]*: *What is the best fitness function in terms of effectiveness?*
  RQ1.2 *[Efficiency]*: *What is the best fitness function in terms of efficiency?*

- ***RQ2****: What is the best solver for detecting XMLi vulnerabilities?* In this second research question, we compare to what extent different search algorithms are able to detect XMLi vulnerabilities when optimizing the same fitness function (e.g., the string edit distance). Specifically, we compare the different algorithms discussed in Section 3.2.3 when optimizing the same fitness function with respect to their ability to detect as many XMLi vulnerabilities as possible (effectiveness) and the time needed to detect such vulnerabilities (efficiency). Therefore, we consider the following two sub-questions:

  RQ2.1 *[Effectiveness]: What is the best solver in terms of effectiveness?*
  RQ2.2 *[Efficiency]: What is the best solver in terms of efficiency?*

The goal of these two research questions is to understand which solver and fitness function combination is more effective and efficient for detecting XMLi vulnerabilities. Therefore, to answer them, we use all web applications in Study 1 to perform an extensive analysis of all possible combinations of solvers and fitness functions (see Section 4.3).

For the industrial applications (M and R) in Study 2, we could not involve our industrial partners in the evaluation of all possible configurations given the high computational cost of this type of study. Indeed, such a detailed investigation involves (i) different solvers, (ii) different fitness functions, (iii) different configurations, (iv) various TOs for each application, and (v) a number of repetitions to address the randomized nature of the solvers being compared. For these reasons, Study 2 is used to evaluate the applicability of the best configuration of our search-based approach, in a realistic context, as formulated by the following research question:

- ***RQ3****: How does the proposed technique perform on large and complex industrial systems?* For this research question, we focus on the two real-world applications M and R in Study 2 to understand whether the proposed search-based approach is able to detect XMLi vulnerabilities (effectiveness) in larger systems with complex input validation routines and in a reasonable amount of time (efficiency). The goal here is to assess the scalability of our approach with real-world industrial systems where the response time is typically slow due to the interactions of several components and/or the complexity of operations. Since running all combinations of solvers and fitness functions is not possible on our industrial applications due to higher execution times, we focus on assessing the best combination of solver and fitness function identified when answering **RQ1** and **RQ2**.

4.3 Variable Selection

To answer our RQs, we studied the effect of the following independent variables:

- *Fitness function*: in Section 3.2.2 we described two different fitness functions, i.e., *real-coded edit distance (Rd)* and *string edit distance (Ed)*, that can be used to guide search algorithms toward the detection of XMLi vulnerabilities. The former has been widely applied in the software testing literature [15] while the latter has been introduced in this paper. To answer **RQ1**, we

compare the results achieved by each solver considered individually when optimizing the two fitness functions for each application and TO in our empirical study.

- *Solver*: given a fitness function, different optimization algorithms can be used to find optimal solutions to our problem. Therefore, this independent variable accounts for the four solvers described in Section 3.2.3 that could be used interchangeably for the XMLi vulnerabilities detection problem, which are Random Search (RS), Hill Climbing (HC), Standard Genetic Algorithms (SGA) and Real-coded Genetic Algorithms (RGA). To answer **RQ2**, we compare the four solvers when optimizing the same fitness function. In other words, the comparison is performed by considering each fitness function separately.

For brevity, in the following we refer to combinations of these two independent variables (*Fitness function* $\times$ *Solver*) as *treatments* affecting the dependent variables.

In our study, the dependent variables are the performance metrics used to compare the effectiveness and the efficiency across treatments. For effectiveness, we use the *Success Rate*, which is the ratio of the number of times a given TO is covered by a treatment Ω to the total number of times the treatment Ω is executed (i.e., runs). More formally, the success rate is defined as follows:

$$SR(TO, \Omega) = \frac{\#\ successful\_runs}{\#\ runs} \times 100 \qquad (12)$$

where # *successful_runs* denotes the number of times Ω covers the TO, and # *runs* indicates the total number of runs.

For efficiency, we use the *Execution Time*, which measures the average time (in minutes) taken by a treatment Ω to reach the termination criterion (i.e., either the TO is covered or the search timeout is reached) over the total number of runs for a given TO.

In addition to the dependent and independent variables described above, we also investigate the following co-factors that may affect the effectiveness and the efficiency across the treatments:

- *Number of input parameters*: each web application in our studies is a web-form with different input boxes where attackers can introduce malicious input strings. A higher number of input strings may increase the search time required by a given treatment to cover each TO. Therefore, we investigate the effect of this co-factor by applying each treatment on subjects with different number of input parameters. The purpose of this analysis is to measure the effect of increasing the number of input parameters on each treatment.
- *Alphabet size*: the alphabet of characters to use for generating input strings is represented by the set of printable ASCII characters. Instead of using the complete alphabet of all possible characters, we can reduce the size of the alphabet for the input parameters by omitting the characters, we know, are unused in the TOs. For example, if we observe that the target TO does not contain the character "`A`", we can assume that such a character is not useful to create malicious input strings. Therefore, we can reduce the size of the search space by removing the character "`A`" from the set of characters (alphabet) to use for generating malicious input. On the other hand, it may be difficult to determine what the restricted alphabet is when data validation and transformation routines are used. Therefore, we assess to what extent the usage of a restricted alphabet (positively/negatively) impacts the performance of search algorithms and fitness functions.
- *Initial population*: all solvers start with an initial set of randomly generated solutions, which are tuples of randomly generated strings. Since the length of the input string that matches the target TO (upon the generation of the corresponding XML message) is unknown a priori, the length of the input strings in the initial population may affect the performance of our treatments. Indeed, if the randomly generated input strings are too long or too short (compared to the final solution) we would expect that each treatment will require more time (more edit operations) to find the malicious input string. To analyze the impact of the initial population on the performance of our treatments, we consider two different settings: (i) we generate random strings with a fixed (F) maximum lengths of characters each, or (ii) we generate strings of variable length (V) by using the "empty" character (see Section 3.2.1).

To perform a detailed evaluation of the effect of these three co-factors on our main treatments, we conducted a number of experiments with different settings, as summarized in Table 2. Each row in the table represents one experiment. The first column contains the name of the applications used in our case studies. The second column (*ExpId*) assigns a unique id to each experiment based on the application and its configuration. The third column #*TOs* lists the number of TOs in the experiment. The fourth column (#*Inp*) lists the number of input parameters. The fifth column (*PopLen.*) reports whether the length of the input strings in the initial population is fixed (Fix) or not (Var). The last column (*Res. Alph.*) indicates whether the search use a full alphabet set (Y) or restricted alphabet set (N). These configuration details are encoded in the *ExpId* values reported in the second column of Table 2. For example, the *ExpId* "S.2.F.Y" encodes the following settings for the SBANK ("S") web application: it has two input parameters ("2"), input strings in the initial population have a fixed length ("F"), and a restricted alphabet set ("Y") is used.

Therefore, we have (3 input parameters $\times$ 2 alphabets $\times$ 2 input string's lengths =) 12 different configurations for both SBANK and SSBANK. Instead, for XMLMao, we have only four possible configurations since this application has only one input parameter. For the industrial applications (M and R) in Study 2, we could not involve our industrial partners in the evaluation of all possible configurations given the high computational cost of this type of study (see Section 4.2). For these reasons, the M and R applications are evaluated with only one configuration, M.2.V.Y and R.3.V.Y respectively. It is worth noticing that the number of input parameters for M and R is fixed since no alternative versions with a different number of input parameters are available. For the remaining setting, we opted for the configurations we empirically found to be statistically superior in Study 1. Therefore, for Study 2 we used the restricted alphabet ("Y") and the initial population composed by input

TABLE 2
Experiment Settings: Experiment ID (Exp. ID) is named based on the corresponding application (App.), the number of inputs (#Inp.), length of input strings in the initial population (PopLen), and whether the alphabet is restricted (Res. Alph.).

App.	Exp. ID	#TOs	#Inp.	PopLen.	Res. Alph.
SBank	S.1.F.N	9	1	Fix	N
	S.2.F.N	10	2	Fix	N
	S.3.F.N	10	3	Fix	N
	S.1.F.Y	9	1	Fix	Y
	S.2.F.Y	10	2	Fix	Y
	S.3.F.Y	10	3	Fix	Y
	S.1.V.N	9	1	Var	N
	S.2.V.N	10	2	Var	N
	S.3.V.N	10	3	Var	N
	S.1.V.Y	9	1	Var	Y
	S.2.V.Y	10	2	Var	Y
	S.3.V.Y	10	3	Var	Y
SSBank	SS.1.F.N	9	1	Fix	N
	SS.2.F.N	10	2	Fix	N
	SS.3.F.N	10	3	Fix	N
	SS.1.F.Y	9	1	Fix	Y
	SS.2.F.Y	10	2	Fix	Y
	SS.3.F.Y	10	3	Fix	Y
	SS.1.V.N	9	1	Var	N
	SS.2.V.N	10	2	Var	N
	SS.3.V.N	10	3	Var	N
	SS.1.V.Y	9	1	Var	Y
	SS.2.V.Y	10	2	Var	Y
	SS.3.V.Y	10	3	Var	Y
XMLMao	X.1.F.N	12	1	Fix	N
	X.1.F.Y	12	1	Fix	Y
	X.1.V.N	12	1	Var	N
	X.1.V.Y	12	1	Var	Y
M	M.2.V.Y	4	2	Var	Y
R	R.3.V.Y	4	3	Var	Y

strings with variable length ("V").

4.4 Experimental protocol

For each TO and each configuration, we executed each treatment Ω and recorded whether the TO is covered or not as well as the execution time. Each execution (i.e., run) is repeated 10 times (but only three times for the industrial systems) to account for the randomized nature of the optimization algorithms. The coverage data is binary since a given TO is either covered or not by a specific run of the treatment, whereas the execution time is recorded in minutes. This data is further used to calculate the selected performance metrics, i.e., *Success Rate* and average *Execution Time* for each TO.

For answering **RQ1.1**, we analyzed whether the success rates achieved by the solvers statistically differ when using two different fitness functions, i.e., real-coded edit distance (Rd) and the string edit distance (Ed). To this aim, we use the Fisher's exact test [29] with a level of significance α = 0.05. The Fisher exact test is a parametric test for statistical significance and is well-suited to test differences between ratios, such as the percentage of times a TO is covered. When the p-value is equal or lower than α, the null hypothesis can be rejected in favor of the alternative one, i.e., a solver (e.g., HC) with one fitness function (e.g., Rd) covers the TO more frequently than the same solver but with another fitness function (e.g., Ed). We also use the Odds Ratio (OR) [30] as measure of the effect size, i.e., the magnitude of the difference between the success rates achieved by Rd and Ed. The higher the OR, the higher is the magnitude of the differences. When the Odds Ratio is equal to 1, the two treatments being compared have the same success rate. Alternatively, OR >1 indicates that the first treatment achieves a higher success rate than the second one and OR <1 the opposite case.

For answering **RQ1.2**, we analyzed whether the execution time achieved by the solvers statistically differs when using Rd or Ed. To compare execution times, we use the non-parametric Wilcoxon test [31] with a level of significance α = 0.05. When obtaining p-values $\leq \alpha$, we can reject the null hypothesis, i.e., a given treatment takes less time to cover the TO under analysis than another treatment. We also use the Vargha-Delaney ($\hat{A}_{12}$) statistic [32] to measure the magnitude of the difference in the execution time. A value of 0.5 for the $\hat{A}_{12}$ statistics indicates that the first treatment is equivalent, in terms of execution time to the second one. When the first treatment is better (lower execution time) than the second one, $\hat{A}_{12} < 0.5$. Naturally, $\hat{A}_{12} > 0.5$ otherwise.

For **RQ2.1** and **RQ2.2**, we use the Friedman's test [33] to verify whether multiple treatments are statistically different or not. It is a non-parametric equivalent to the ANOVA test [34] and thus does not make any assumption about the data distributions to be compared. More specifically, for **RQ2.1**, we compare the average success rates achieved by the different treatments in 10 independent runs across all web applications and configurations. Instead, for **RQ2.2** the comparison is performed considering the average execution time achieved in the 10 runs across all web applications and configurations. For both **RQ2.1** and **RQ2.2**, we use a level of significance α = 0.05. When the p-values obtained from the Friedman's test are significant (i.e., <= 0.05), we apply the post-hoc Conover's procedure [35] for pairwise multiple comparison. The p-values produced by the post-hoc Conover's procedure are further adjusted using the Holm-Bonferroni procedure [36] to correct the significance level in case of multiple comparisons. Note that the purpose of **RQ2.1** and **RQ2.2** is to compare different solvers in terms of both effectiveness and efficiency; thus, we separately compare the four solvers described in Section 3.2.3 for the two fitness functions (e.g., Rd and Ed).

4.5 Parameter settings

Running randomized algorithms, and GAs in particular, requires to set various parameters to achieve acceptable results. In this study, we set the parameter values by following the recommendations in the related literature, as detailed below:

- **Mutation rate**. De Jong's [37] recommended value of p_m=0.001 for mutation rate has been used by many implementations of Genetic Algorithms. Another popular mutation rate has been defined by Grefenstette's [38] as p_m=0.01. Further studies [39], [40], [41], [42] have demonstrated that p_m values based on the population size and chromosome's length achieves better performance. Hence, for RGA and SGA we use p_m =

$(1.75)/(\lambda\sqrt{l})$ as mutation rate, where l is the length of the chromosome and λ is the population size. We also conducted some preliminary experiments with these different recommended mutation rates and we found that better results are indeed achieved when p_m is based on the population size and chromosome's length. For HC, we set the mutation rate to $1/l$ (where l is the length of the chromosome) since there is no population for this solver. This parameter is not applicable for RS.

- **Crossover rate**. The crossover rate is another important factor for the performance of GAs. The recommended range for the crossover rate is $0.45 \leq p_c \leq 0.95$ [39], [43]. In our experiments, we chose $p_c = 0.70$ for RGA/SGA, which falls within the range of the recommended values. Notice that this parameter is not applicable to HC and RS.
- **Population size**. Selecting a suitable population size for GAs is also a challenging task since it can affect their performance. The recommended values used in the literature are within the range 30-80 [43]. From our preliminary experiments, we observed that the population size of 50 works best for RGA/SGA in our context. Such a value is also consistent with the parameters settings used in recent studies in search-based software testing [44], [45], [46]. This parameter is applicable only to population-based algorithms, i.e., RGA and SGA in our case.
- **Termination Criteria**. The search terminates when one of the following two stopping criteria is satisfied: a zero-fitness value is obtained (i.e., the target TO is covered) or the maximum number of fitness evaluations is reached. For SBANK, XMLMao and the two industrial systems, we set the maximum number of fitness evaluations to 300K. Instead, for SSBANK, we used a larger search budget of 500K fitness evaluations because it uses input validation routines, which make the TOs more difficult to cover. We also empirically found that a larger search budget is indeed needed for SSBANK compared to SBANK and XMLMao.

4.6 Implementation

We have implemented all solvers and all fitness functions in a prototype tool implemented on top of JMetal [47], which is an optimization framework written in Java. The tool takes as inputs the SUT, the TOs containing malicious *XMLi* content, the solver to apply, and the fitness function to optimize. The tool generates test cases according to the given combination of solver/fitness function, i.e., inputs for the SUT (e.g., input values in HTML forms) that lead to *XMLi* attacks. Each candidate test is evaluated by executing the SUT with the corresponding inputs and comparing the generated XML message with the target TO. The comparison is based on the selected fitness function while tests are evolved according to the selected solver.

The tool is composed of two main components: (i) the test case generator, and (ii) the test executor. The *test case generator* is the core component of the tool and it is implemented on top of jMetal [47]. This component implements the search algorithms (i.e., RS, HC, SGA, and RGA) and the fitness functions (i.e., the String Edit Distance and the Real-Coded Edit Distance) described in Section 3. The *test executor* provides an interface between the SUT and the *test case generator*. It takes the input strings generated by the *test case generator* and submits them to the SUT (e.g, through a HTTP POST). The XML messages produced by the SUT for the web services are intercepted and forwarded to the *test case generator*, which calculates the corresponding fitness function scores. In other words, the *test executor* is an HTTP proxy between the SUT and the web services.

Each SUT requires its own test executor to correctly interact with the user interface (e.g., HTML web forms and input parameter names) and to intercept the generated XML files (e.g., SOAP messages or data bodies in HTTP POST messages toward RESTful web services). However, we use the same test executor for all solvers and fitness functions when testing the same SUT.

It is worth noticing that all solvers and fitness functions are implemented in the same tool, using the same programming language (i.e., Java) and relying on the search operators (e.g., mutation) available in JMetal. This setting avoids potential confounding factors due to the usage of different tools with different implementation details when measuring the execution time of the different solvers.

5 Results

This section discusses the results of our case studies, addressing in turn each of the research questions formulated in Section 4. Reporting the individual results along with the statistical tests for each TO, for each configuration, and for each treatment is not feasible due to the large number of resulting combinations, i.e., 2,016 in total. Therefore, we report the mean and standard deviation of the success rate and of the execution time obtained for all TOs of the same web application and with the same configuration (i.e., for each experiment/row in Table 2). For the statistical tests, we report the number of times the differences between pairs of treatments are statistically significant together with the average effect size measures.

5.1 RQ1: What is the best fitness function for detecting XMLi vulnerabilities?

Table 3 summarizes the results of all treatments on the first subject SBANK in Study 1, listing the average success rate (SR) along with the standard deviation (SD) for each configuration. Each row in the table represents one configuration (experiment) identified by the unique id listed in the first column $ExpId$. The last row in the table lists the mean values for SR and SD across all configurations. As depicted in that table, for all solvers the real-coded edit distance (Rd) achieved higher success rates compared to the string edit distance (Ed). RGA achieved an SR of 95.83% with Rd, which is much higher than that of Ed with 16.07%.

These observations are confirmed by the Fisher's exact test, as reported in Table 4. For each solver, this table lists the average Odds Ratios (OR) of the success rates for each configuration, as well as the number of times where Rd achieved significantly higher ($\#Rd > Ed$) or lower ($\#Rd < Ed$) success rates compared to Ed, according to the Fisher's exact test. The last row in the table lists (i)

TABLE 3
Average Success Rates (SR) and Standard Deviation (SD) out of 10 runs per TO for SBANK

ExpId	RGA				SGA				HC				RS	
	Rd		Ed		Rd		Ed		Rd		Ed		Ed	
	SR	SD	SR	SD	SR	SD	SR	SD	SR	SD	SR	SD	SR	SD
S.1.F.N	100.00	0.00	22.22	23.86	71.11	38.87	67.78	29.49	92.22	13.02	90.00	7.07	0.00	0.00
S.2.F.N	96.00	6.99	8.00	11.35	50.00	43.97	39.00	29.61	88.00	22.01	79.00	23.31	0.00	0.00
S.3.F.N	93.00	10.59	3.00	6.75	40.00	34.96	28.00	27.81	60.00	51.64	45.00	41.16	0.00	0.00
S.1.F.Y	100.00	0.00	35.56	33.21	85.56	10.14	73.33	17.32	98.89	3.33	90.00	11.18	0.00	0.00
S.2.F.Y	92.00	7.89	21.00	20.25	45.00	28.38	44.00	36.88	89.00	14.49	90.00	10.54	0.00	0.00
S.3.F.Y	87.00	19.47	7.00	9.49	41.00	33.48	26.00	21.71	60.00	51.64	47.00	42.96	0.00	0.00
S.1.V.Y	100.00	0.00	40.00	36.40	100.00	0.00	78.89	13.64	97.78	4.41	86.67	14.14	0.00	0.00
S.2.V.Y	99.00	3.16	26.00	27.16	75.00	35.36	56.00	25.03	71.00	41.75	58.00	26.58	0.00	0.00
S.3.V.Y	93.00	13.37	6.00	8.43	69.00	41.75	46.00	26.75	70.00	48.30	37.00	34.66	0.00	0.00
S.1.V.N	100.00	0.00	21.11	27.13	77.78	33.46	67.78	27.74	96.67	7.07	86.67	11.18	0.00	0.00
S.2.V.N	100.00	0.00	3.00	6.75	61.00	50.43	41.00	35.10	76.00	35.02	63.00	32.34	0.00	0.00
S.3.V.N	90.00	15.63	0.00	0.00	55.00	47.90	24.00	23.19	60.00	51.64	29.00	29.98	0.00	0.00
Average	95.83	6.43	16.07	17.57	64.20	33.23	49.31	26.19	79.96	28.69	66.78	23.76	0.00	0.00

TABLE 4
Average Odds Ratios (OR) of the Success Rate for SBANK application. For each solver, we also report the number of times the Success Rate obtained by the real-coded distance is statistically better (# Rd $>$ Ed) or worse (# Rd $<$ Ed) than the edit distance.

ExpId	RGA			SGA			HC		
	Avg. OR	#Rd$>$ Ed	#Rd$<$Ed	Avg. OR	#Rd$>$Ed	#Rd$<$Ed	Avg. OR	#Rd$>$Ed	#Rd$<$Ed
S.1.F.N	216.25	8	0	2.88	0	0	2.30	0	0
S.2.F.N	223.93	10	0	2.55	0	0	2.36	0	0
S.3.F.N	235.08	10	0	2.10	0	0	5.98	1	0
S.1.F.Y	164.75	6	0	3.14	0	0	3.57	0	0
S.2.F.Y	57.59	10	0	2.13	0	0	2.39	0	0
S.3.F.Y	118.90	9	0	2.44	0	0	5.28	1	0
S.1.V.Y	162.06	5	0	7.16	0	0	3.81	0	0
S.2.V.Y	167.45	8	0	6.84	0	0	7.12	1	0
S.3.V.Y	196.65	10	0	8.79	1	0	54.14	3	0
S.1.V.N	224.83	8	0	3.59	0	0	3.22	0	0
S.2.V.N	373.24	10	0	8.08	1	0	5.25	1	0
S.3.V.N	269.64	10	0	10.36	3	0	22.89	4	0
Avg./Total	200.87	104	0	5.00	5	0	9.86	11	0

TABLE 5
Average Success Rates (SR) and Standard Deviation (SD) out of 10 runs per TO for SSBANK

ExpId	RGA				SGA				HC				RS	
	Rd		Ed		Rd		Ed		Rd		Ed		Ed	
	SR	SD	SR	SD	SR	SD	SR	SD	SR	SD	SR	SD	SR	SD
S.1.F.N	66.67	50.00	17.78	24.38	63.33	47.70	52.22	41.77	62.22	47.11	56.67	43.01	0.00	0.00
S.2.F.N	3.00	4.83	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
S.3.F.N	3.00	6.75	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
S.1.F.Y	66.67	50.00	23.33	28.28	56.67	43.59	50.00	39.05	63.33	48.48	57.78	44.10	0.00	0.00
S.2.F.Y	6.00	5.16	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
S.3.F.Y	7.00	6.75	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
S.1.V.Y	66.67	50.00	32.22	39.62	64.44	48.51	54.44	42.75	64.44	48.51	47.78	37.34	0.00	0.00
S.2.V.Y	18.00	15.49	2.00	4.22	12.00	16.19	5.00	7.07	0.00	0.00	0.00	0.00	0.00	0.00
S.3.V.Y	21.00	19.12	2.00	4.22	6.00	12.65	3.00	4.83	0.00	0.00	0.00	0.00	0.00	0.00
S.1.V.N	66.67	50.00	13.33	20.62	62.22	47.64	46.67	37.42	65.56	49.27	44.44	37.45	0.00	0.00
S.2.V.N	4.00	6.99	0.00	0.00	2.00	4.22	2.00	4.22	0.00	0.00	0.00	0.00	0.00	0.00
S.3.V.N	5.00	9.72	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Average	27.81	22.90	7.56	10.11	22.22	18.37	17.78	14.76	21.30	16.11	17.22	13.49	0.00	0.00

the average OR for all configurations (i.e., average of the columns $Avg.\ OR$), and (ii) the total number of statistically significant cases (i.e., the sum of the $\#Rd > Ed$ / $\#Rd < Ed$ columns). We can observe that, for all solvers and for all configurations, the OR is always larger than one. The largest OR values are obtained for RGA, for which we observe that Rd is significantly better than Ed in most of the configurations ($\approx$ 90%), with an average OR value ranging between 57.59 to 373.24. For the other solvers, OR is still larger than one but its magnitude is smaller when compared to that of RGA. In addition, according to the Fisher's exact test, SGA and HC performed significantly better with Rd in only 4% and 9% of the configurations, respectively. These results indicate that the solver that most benefits from the usage of Rd is RGA.

For SSBANK in Study 1, success rate results are listed in Table 5. Despite the input validations in SSBANK, the

TABLE 6
Average Odds Ratios (OR) of the Success Rate for SSBANK application. For each solver, we also report the number of times the Success Rate obtained by the real-coded distance is statistically better (# Rd $>$ Ed) or worse (# Rd $<$ Ed) than the edit distance.

ExpId	RGA			SGA			HC		
	Avg. OR	#Rd$>$ Ed	#Rd$<$Ed	Avg. OR	#Rd$>$Ed	#Rd$<$Ed	Avg. OR	#Rd$>$Ed	#Rd$<$Ed
S.1.F.N	114.42	5	0	3.04	0	0	2.02	0	0
S.2.F.N	1.69	0	0	1.00	0	0	1.00	0	0
S.3.F.N	1.75	0	0	1.00	0	0	1.00	0	0
S.1.F.Y	107.27	4	0	2.12	0	0	2.71	0	0
S.2.F.Y	2.39	0	0	1.00	0	0	1.00	0	0
S.3.F.Y	2.68	0	0	1.00	0	0	1.00	0	0
S.1.V.Y	102.81	3	0	4.31	1	0	5.47	1	0
S.2.V.Y	4.91	0	0	2.29	0	0	1.00	0	0
S.3.V.Y	5.78	0	0	1.34	0	0	1.00	0	0
S.1.V.N	158.04	6	0	6.57	1	0	7.55	1	0
S.2.V.N	1.98	0	0	1.16	0	0	1.00	0	0
S.3.V.N	2.34	0	0	1.00	0	0	1.00	0	0
Avg./Total	42.17	18	0	2.15	2	0	2.15	2	0

TABLE 7
Average Success Rates (SR) and Standard Deviation (SD) out of 10 runs per TO for XMLMao

ExpId	RGA				SGA				HC				RS	
	Rd		Ed		Rd		Ed		Rd		Ed		Ed	
	SR	SD	SR	SD	SR	SD	SR	SD	SR	SD	SR	SD	SR	SD
X.1.F.N	100.00	0.00	35.00	38.26	78.33	24.80	70.83	29.37	98.33	3.89	91.67	8.35	0.00	0.00
X.1.F.Y	100.00	0.00	44.17	44.41	70.83	19.75	70.83	23.92	96.67	6.51	91.67	12.67	0.00	0.00
X.1.V.Y	100.00	0.00	51.67	37.86	95.83	11.65	82.50	16.03	95.00	6.74	90.00	11.28	0.00	0.00
X.1.V.N	100.00	0.00	30.00	32.19	85.83	22.75	81.67	21.25	95.00	6.74	88.33	10.30	0.00	0.00
Average	100.00	0.00	40.21	38.18	82.71	19.74	76.46	22.64	96.25	5.97	90.42	10.65	0.00	0.00

TABLE 8
Average Odds Ratios (OR) of the Success Rate for XMLMao application. For each solver, we also report the number of times the Success Rate obtained by the real-coded distance is statistically better (# Rd $>$ Ed) or worse (# Rd $<$ Ed) than the edit distance.

ExpId	RGA			SGA			HC		
	Avg. OR	#Rd$>$ Ed	#Rd$<$Ed	Avg. OR	#Rd$>$Ed	#Rd$<$Ed	Avg. OR	#Rd$>$Ed	#Rd$<$Ed
X.1.F.N	175.32	8	0	2.50	0	0	2.81	0	0
X.1.F.Y	167.90	7	0	2.06	0	0	2.89	0	0
X.1.V.Y	94.09	6	0	4.94	0	0	2.06	0	0
X.1.V.N	156.95	9	0	2.51	0	0	2.32	0	0
Avg./Total	148.56	30	0	3.00	0	0	2.52	0	0

solvers were able to obtain positive success rates in many configurations with both Rd and Ed. This means that some *XMLi* attacks can be still generated by inserting malicious inputs. Thus, the input validation procedures in SSBANK is sub-optimal, either because it is not adequately implemented, or because it is not possible to avoid all possible attacks using only input validation. We further observe that Rd achieved higher average success rates compared to Ed in all configurations. Indeed, the average improvement of the success rate when using Rd is 20% for RGA, 4% for SGA and 3% for HC. The corresponding results of the Fisher's exact test and OR values are provided in Table 6. Similar to the results achieved for SBANK, OR is larger than one in most of the configurations, although the larger differences are observed for RGA. In particular, for this solver, Rd leads to an average OR ranging between 1.69 and 107.27. Instead, for the other two solvers, there is no statistically significant difference according to Fisher's exact test for most of the cases, as confirmed by the OR values which are often around one.

For XMLMao in Study 1, the results for average success rates are provided in Table 7. When applying RGA with Rd, the success rate is 100% for all configurations, which is much higher than that of Ed with 40%. This large difference is also confirmed by the Fisher's exact test and very large OR values. Indeed, RGA with Rd is significantly better than RGA with Ed in 30 cases out of 48 (62%). The corresponding average OR values are very large, ranging between 94.09 and 175.32. In contrast, for the other two solvers, the differences are never significant when comparing the two fitness functions.

In general, for all three subjects in Study 1, we observe that Random Search (RS) always results in zero success rate, i.e., it was unable to cover any TO. This confirms the need for more advanced search algorithms to detect XMLi vulnerabilities. Furthermore, none of the solvers reached significantly higher success rates when using Ed instead of Rd. Therefore, for **RQ1.1**, we conclude that:

The real-coded edit distance is very effective compared to the string edit distance, especially for RGA which, as shown next, is the best solver as well.

TABLE 9
Average execution time (in minutes) results for SBANK

ExpId	RGA		SGA		HC		RS
	Rd	Ed	Rd	Ed	Rd	Ed	
S.1.F.N	2.26	8.82	9.27	5.21	4.37	3.65	8.52
S.2.F.N	5.21	10.06	13.55	7.75	8.49	6.48	8.82
S.3.F.N	6.51	8.74	14.26	8.24	8.91	8.53	8.40
S.1.F.Y	1.52	7.19	7.13	4.73	2.20	2.33	9.11
S.2.F.Y	4.38	8.30	11.82	6.93	8.24	3.95	10.32
S.3.F.Y	5.73	8.03	12.28	7.83	9.52	7.42	9.05
S.1.V.Y	1.51	6.83	5.08	4.14	2.08	2.46	7.87
S.2.V.Y	2.95	8.14	8.92	6.14	12.10	5.65	7.90
S.3.V.Y	4.98	8.38	10.42	6.88	7.58	7.28	7.47
S.1.V.N	2.17	7.89	8.98	5.71	3.53	3.16	10.93
S.2.V.N	4.21	9.21	11.84	7.71	7.95	5.72	12.03
S.3.V.N	6.08	7.83	12.36	7.77	8.40	7.59	10.33
Average	3.96	8.28	10.49	6.59	6.95	5.35	9.23

TABLE 10
Average execution time (in minutes) results for SSBANK

ExpId	RGA		SGA		HC		RS
	Rd	Ed	Rd	Ed	Rd	Ed	
S.1.F.N	5.12	15.21	11.45	8.20	7.09	8.18	12.95
S.2.F.N	10.50	6.87	10.11	8.57	11.92	13.46	5.36
S.3.F.N	5.33	3.87	6.06	4.95	6.76	5.47	1.70
S.1.F.Y	5.49	13.88	8.77	7.96	6.67	6.11	13.86
S.2.F.Y	11.62	6.99	9.73	9.23	12.81	10.72	5.77
S.3.F.Y	6.24	4.15	5.94	4.29	7.26	4.78	1.71
S.1.V.Y	5.69	14.20	8.75	9.03	6.00	8.63	11.49
S.2.V.Y	11.40	8.89	11.42	10.41	12.48	11.60	4.85
S.3.V.Y	6.41	3.98	8.32	5.61	6.29	4.84	1.68
S.1.V.N	5.57	14.49	10.65	11.62	7.14	8.87	13.08
S.2.V.N	11.52	5.80	9.44	8.81	12.07	9.92	5.49
S.3.V.N	5.49	4.24	6.20	4.14	6.16	5.35	1.66
Average	7.53	8.55	8.90	7.74	8.56	8.16	6.63

Regarding efficiency (**RQ1.2**), Tables 9, 10 and 11 report the average execution time for the three subjects SBANK, SSBANK and XMLMao in Study 1, respectively. The results of the Wilcoxon's test, along with the $\hat{A}_{12}$ statistics, are reported in Tables 12, 13 and 14. For each solver and each configuration, these tables list the effect size as well as the number of cases where the execution time for Rd is significantly lower or higher than Ed, based on the Wilcoxon's test and $\hat{A}_{12}$ statistics.

Unlike the results of the success rates where Rd always performed better, we obtained mixed results for different solvers and applications when looking at efficiency. For SBANK, RGA with Rd exhibited better efficiency with an average execution time of 3.96 minutes compared to 8.28 minutes for Ed. This is also confirmed by the reported low $\hat{A}_{12}$ values (e.g., 0.14) and a significantly more efficient Rd in 80% of the cases. For SGA and HC, Ed obtained lower execution times and is significantly more efficient than Rd in 50% and 39% of the cases, respectively. Efficiency results for XMLMao are similar to SBANK except for HC, for which the average execution time obtained with Rd is lower and is found to be significantly better than Ed in seven cases. Regarding SSBANK, the differences in average execution time obtained with Rd and Ed are not very large (i.e., $\approx$ 1 minute), although statistically significant in favor of Ed in many cases, i.e., 25-53% for $\#Rd > \#Ed$.

Overall, in terms of efficiency, the real-coded edit distance is significantly better than the string edit distance for RGA, while the reverse is true for SGA and HC. One possible explanation for this difference is the better ability of the genetic operators in RGA to exploit the neighborhood of candidate solutions when using Rd. As explained in Section 3, Rd helps focus on sub-regions of the search space but it is necessary that the solvers are able to exploit this information to produce some benefits. To better explain this aspect, let us consider the TO=`A` (ASCII code 65) and let assume that the current input string is `C` (ASCII code 67), whose real-coded edit distance to the TO is $|65-67|/(|65-67|+1) = 0.67$. When using the mutation operators of HC and SGA, the character `C` can be replaced by any other character with ASCII code from 32 to 127 even if only few characters in this set would lead to better Rd values, i.e., those with ASCII codes $\in \{64, 65, 66\}$. Therefore, the probability of replacing the character `C` with a better character is very low, i.e., $p = 3/95 \approx 0.03$. Instead, in RGA the gaussian mutation gives higher probability to characters with ASCII codes that are closer to 67, which is the code of `C`. Indeed, the probability of replacing `C` with characters with ASCII codes $\in \{64, 65, 66\}$ is much higher in RGA when compared to HC and SGA. On the other hand, Rd is more expensive to compute than Ed since it is based on real-numbers and entails additional computations (as shown in Equation 2 in Section 3). Therefore, Rd will lead to better efficiency if and only if its additional overhead is compensated by a large saving in the number of fitness evaluations.

TABLE 11
Average execution time (in minutes) results for XMLMao

ExpId	RGA		SGA		HC		RS
	Rd	Ed	Rd	Ed	Rd	Ed	
X.1.F.N	1.04	8.02	7.18	5.98	1.86	3.21	12.19
X.1.F.Y	1.02	6.73	7.45	6.17	1.35	2.09	8.84
X.1.V.Y	0.81	6.44	4.76	5.41	1.38	2.29	12.11
X.1.V.N	0.90	8.33	6.86	5.92	2.34	3.77	8.64
Average	0.94	7.38	6.56	5.87	1.73	2.84	10.44

After manual investigation, we discovered that this is the case only for RGA. Indeed, Rd remained efficient for RGA in most of the cases due to a higher success rate than Ed, which resulted in a lower number of fitness evaluations during search. Instead, for SGA and HC the reduction in the number of fitness evaluations is small and thus it does not compensate for the additional overhead of Rd with respect to Ed. The only exception to this general rule is SSBANK, for which RGA with Rd is both more effective and less efficient than Ed. These results are due to the input validations performed in SSBANK, which produces an error message instead of a complete XML response whenever invalid inputs are submitted. When using Ed, computing the distance between such a small error message and the TO is much faster than doing so with a complete XML output generated upon the insertion of valid inputs.

In other words, our investigation reveals that the real-coded edit distance is more efficient, in terms of execution time, in the specific case where it achieves a much higher success rate than the string edit distance. Otherwise, if the success rates of the two fitness functions do not differ

TABLE 12
Average A_{12} statistics of the execution time for SBANK application. For each solver, we also report the number of times the efficiency of the real-coded distance is statistically better (# Rd $>$ Ed) or worse (# Rd $<$ Ed) than the edit distance.

ExpId	RGA			SGA			HC		
	Eff.Siz	#Rd>Ed	#Rd<Ed	Eff.Siz	#Rd>Ed	#Rd<Ed	Eff.Siz	#Rd>Ed	#Rd<Ed
S.1.F.N	0.05	8	0	0.81	0	5	0.62	0	3
S.2.F.N	0.11	8	0	0.79	0	4	0.77	0	5
S.3.F.N	0.38	6	4	0.88	0	6	0.60	0	1
S.1.F.Y	0.11	7	0	0.75	0	4	0.61	1	2
S.2.F.Y	0.12	10	0	0.82	0	6	0.62	0	3
S.3.F.Y	0.27	6	0	0.76	0	4	0.70	1	5
S.1.V.Y	0.06	8	0	0.68	1	4	0.61	0	0
S.2.V.Y	0.03	10	0	0.73	0	4	0.60	1	3
S.3.V.Y	0.20	6	0	0.72	0	5	0.65	0	3
S.1.V.N	0.01	9	0	0.81	0	5	0.65	0	3
S.2.V.N	0.04	9	0	0.80	0	5	0.79	0	7
S.3.V.N	0.34	6	2	0.83	0	7	0.59	1	4
Avg./Total	0.14	93	6	0.78	1	59	0.65	4	39

TABLE 13
Average A_{12} statistics of the execution time for SSBANK application. For each solver, we also report the number of times the efficiency of the real-coded distance is statistically better (# Rd $>$ Ed) or worse (# Rd $<$ Ed) than the edit distance.

ExpId	RGA			SGA			HC		
	Eff.Siz	#Rd>Ed	#Rd<Ed	Eff.Siz	#Rd>Ed	#Rd<Ed	Eff.Siz	#Rd>Ed	#Rd<Ed
S.1.F.N	0.12	6	0	0.81	0	6	0.47	1	0
S.2.F.N	0.80	0	8	0.70	0	2	0.49	1	1
S.3.F.N	0.77	0	5	0.72	0	3	0.53	0	2
S.1.F.Y	0.33	6	2	0.59	0	1	0.69	0	2
S.2.F.Y	0.87	0	9	0.55	0	0	0.65	0	3
S.3.F.Y	0.83	0	9	0.80	0	6	0.68	1	6
S.1.V.Y	0.21	6	1	0.54	1	1	0.48	1	1
S.2.V.Y	0.65	0	5	0.53	0	1	0.65	1	2
S.3.V.Y	0.68	0	3	0.69	0	5	0.75	0	4
S.1.V.N	0.28	6	1	0.49	3	2	0.56	0	0
S.2.V.N	0.91	0	9	0.52	0	0	0.67	0	4
S.3.V.N	0.93	0	10	0.75	0	7	0.63	1	5
Avg./Total	0.62	24	62	0.64	4	34	0.60	6	30

TABLE 14
Average A_{12} statistics of the execution time for XMLMao application. For each solver, we also report the number of times the efficiency of the real-coded distance is statistically better (# Rd $>$ Ed) or worse (# Rd $<$ Ed) than the edit distance.

ExpId	RGA			SGA			HC		
	Eff.Siz	#Rd>Ed	#Rd<Ed	Eff.Siz	#Rd>Ed	#Rd<Ed	Eff.Siz	#Rd>Ed	#Rd<Ed
X.1.F.N	0.23	8	1	0.66	0	3	0.41	4	0
X.1.F.Y	0.26	8	1	0.61	0	1	0.45	2	0
X.1.V.Y	0.26	8	2	0.54	0	2	0.48	0	1
X.1.V.N	0.16	8	0	0.64	0	2	0.46	1	0
Avg./Total	0.23	32	4	0.61	0	8	0.45	7	1

significantly, the string edit distance is more efficient.

Regarding **RQ1.2**, we conclude that,

> *Unless a significantly higher success rate is achieved by the real-coded edit distance, the string edit distance leads to a more efficient search.*

To answer **RQ1**, we consider both the results of **RQ1.1** and **RQ1.2**. Rd fares better in terms of effectiveness whereas it is worse regarding efficiency. However, even when Rd leads to higher execution times, the difference with Ed ranges between 0.69 to 3.9 minutes on average, which is of limited practical consequences. Further, this relatively small difference is largely compensated with a much higher ability to detect XMLi vulnerabilities, up to an improvement of 80% in detection rate.

5.1.1 Convergence Analysis of Rd and Ed

In Section 3.2.2, we discussed the theoretical advantages of Rd over the traditional Ed. Our results also provide empirical evidence that Rd is more effective than Ed, especially if used in combination with RGA. To better understand why the real-coded distance is so effective, in this subsection we analyze the convergence of the two fitness functions over time. To this end, we recorded the fitness values of the best individual over the different iterations/generations of the genetic algorithms. Due to space limits, we focus on RGA and compare the value of Rd and Ed throughout its generations for one representative case, i.e., SSBANK with the configuration $S.1.F.N$. In particular, we investigate six TOs including four feasible TOs (i.e., they can be generated from the SUT) and two infeasible ones (i.e., input sanitization and

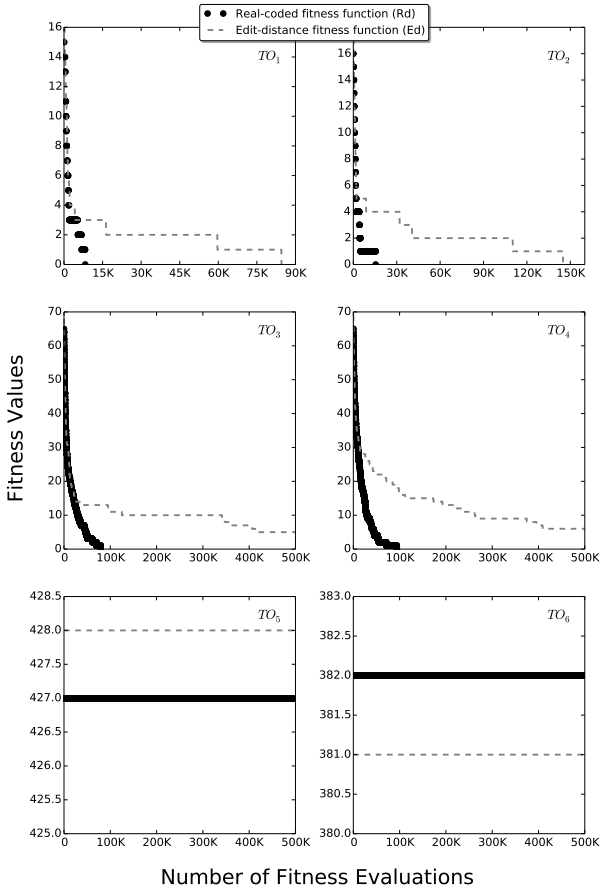


Fig. 6. Convergence rate for RGA with Rd and Ed for SSBANK with $S.1.F.N$ configuration.

validation are able to prevent their generation).

As depicted in Figure 6, RGA with Rd required less than $30K$ fitness evaluations to reach a zero-fitness value for TO1 and TO2, and therefore cover them. Instead, Ed required a larger number of fitness evaluations, i.e., 90K for TO1 and 150K for TO2. RGA with Rd also covered the other two TOs, i.e., TO3 and TO4 in less than 100K fitness evaluations while Ed could not cover them even after 500K fitness evaluations. The last two TOs (TO5 and TO6) are infeasible since the front-end web application in SSBANK contains strong validation routines that always return an error message when the generated XML message becomes too close to TO5 and TO6. For this reason, the fitness function remains flat for both Rd and Ed as depicted in Figure 6, meaning that the search cannot converge towards zero-fitness. For applications with input validation (like SSBANK), it should be expected that some TOs are simply infeasible to cover. However, our results show that it is still possible to detect XMLi vulnerabilities which are not adequately addressed by input validation routines. For these feasible TOs, Rd leads to a faster convergence to a zero-fitness value, thus resulting in better effectiveness (**RQ1.1**) and efficiency (**RQ1.2**).

TABLE 15
Ranking produced by the Friedman's (smaller values of Rank indicate more effectiveness) when using Rd. For each solver, we also report whether it is significantly better than the other solvers according to the post-hoc procedure.

ID	Solver	Rank	Significantly better than
1	RGA	1.02	(2), (3), (4)
2	HC	2.50	(3), (4)
3	SGA	2.70	(4)
4	RS	3.77	-

TABLE 16
Ranking produced by the Friedman's (smaller values of Rank indicate more effectiveness) when using Ed. For each solver, we also report whether it is significantly better than the other solvers according to the post-hoc procedure.

ID	Solver	Rank	Significantly better than
1	HC	1.62	(3), (4)
2	SGA	1.87	(3), (4)
3	RGA	2.86	(4)
4	RS	3.64	-

5.2 RQ2: What is the best solver for detecting XMLi vulnerabilities?

To answer **RQ2.1**, we compare the success rates of the four solvers (i.e., RS, HC, SGA, and RGA) for each fitness function (e.g., Rd and Ed) . As reported in Table 3, the highest success rate (95.83%) for SBANK is achieved by RGA with Rd. Similarly, for SSBANK and XMLMao, RGA with Rd achieved the highest success rates of 27.81% and 100%, respectively. In contrast, the results are mixed when using Ed as fitness function: for SBANK and XMLMao, the highest success rate scores are obtained by HC (66.78% and 90.47% respectively), while for SSBANK the best success rate of 17.78% is obtained by SGA. Finally, RS fares the worst with a success rate of zero in all experiments and subjects, as it could not cover a single TO.

To establish the statistical significance of these results, we use the Friedman's test [33] to compare the average success rates (over ten runs) achieved by the different solvers for all web applications, configuration settings, and TOs. When using Rd as fitness functions, the Friedman's test reveals that the solvers significantly differ from each other in terms of effectiveness (p-value = 2.58×10^{-15}). For completeness, Table 15 provides the ranking obtained by the Friedman's test as well as the results of the post-hoc Conover's procedure [35] for multiple pairwise comparisons. As we can observe, the best rank is obtained by RGA, which turns out to be significantly better than all the other solvers according to the post-hoc Conover's procedure. The four solvers are also significantly different when using Ed as indicated by the Friedman's test, yielding a p-value of 8.2×10^{-12}. However, as visible in Table 16, RGA is not the best solver with this fitness function, being ranked third above RS. The two other solvers, i.e., HC and SGA, are statistically equivalent according to the Conover's tests, though HC obtained a slightly better rank based on Friedman's test.

Given the mixed results obtained for the two fitness functions, we compare the best solver with Rd against

TABLE 17
Ranking produced by the Friedman's (larger values of Rank indicate more efficiency) when using Rd. For each solver, we also report whether it is significantly better than the other solvers according to the post-hoc procedure.

ID	Solver	Rank	Significantly better than
1	RGA	3.53	(2), (3), (4)
2	HC	2.36	(4)
3	RS	2.28	-
4	SGA	1.82	-

the best solver with Ed to find the best treatment (i.e., combination of solvers and fitness functions). To this aim, we performed the Friedman's test comparing the average success rates of RGA with Rd against HC and SGA with Ed. Results show these three treatments are statistically different in terms of effectiveness (p-value = 9.17×10^{-11}). The post-hoc Conover procedure confirms the superiority of RGA with Rd over the other two treatments. Therefore, for **RQ2.1** we conclude that:

RGA combined with the real-coded edit distance as fitness function is the best solver in terms of effectiveness.

Regarding **RQ2.2**, we analyze the execution time of the solvers for each fitness function (i.e., Rd and Ed) separately. The results of this analysis for the three subjects in Study 1 are reported in Tables 9, 10 and 11. For all three subjects, the most efficient solver with Rd is always RGA, whose average running time ranges between 0.94 (for XMLMao) and 7.53 (for SSBANK) minutes. Further, the average execution time for HC ranges between 1.73 (for XMLMao) and 8.56 (for SSBANK) minutes, whereas it ranges between 6.56 and 10.49 minutes for SGA. The differences in execution times are also confirmed by Friedman's test, which returned a p-value of 6.26×10^{-6}. To better understand for which pairs of solvers such a significance holds, Table 17 shows the complete ranking produced by Friedman's test as well as the results of the post-hoc Conover's procedure. The best rank is achieved by RGA, which significantly outperforms all the other solvers when using Rd. HC is ranked second but it is statistically more efficient than SGA only.

When using Ed as fitness function, there is no clear winner among the four solvers in terms of efficiency for the three subjects in Study 1. Indeed, HC is the most efficient solver for XMLMao and SBANK, while SGA is for SSBANK. From the statistical comparison performed with the Friedman's test, we can definitely conclude that the four solvers are significantly different in terms of execution time. However, the post-hoc Conover's procedure revealed that statistical significance holds only when comparing one pair of solvers (see Table 18): HC and RS.

To find out which treatment among all possible combinations of solvers and fitness functions is the most efficient (as measured by the average execution time), we performed Friedman's test to compare RGA with Rd and HC with Ed, which are the best treatments for the two fitness functions (see Tables 17 and Table 18). The resulting p-value of 0.002 and the corresponding Friedman's ranking indicate that RGA with Rd is significantly more efficient than HC with Ed. Thus, addressing **RQ2.2**, we conclude that:

TABLE 18
Ranking produced by the Friedman's (larger values of Rank indicate more efficiency) when using Ed. For each solver, we also report whether it is significantly better than the other solvers according to the post-hoc procedure.

ID	Solver	Rank	Significantly better than
1	HC	3.07	(3)
2	SGA	2.78	-
3	RS	2.21	-
4	RGA	1.93	-

TABLE 19
Results on the industrial systems

Config.	TO	Successes	Avg. Iterations
M.2.V.Y	Close	0	300k
	Meta	0	300k
	Replicate	0	300k
	Replace	3	23k
R.3.V.Y	Close	0	300k
	Meta	0	300k
	Replicate	0	300k
	Replace	2	147k

RGA combined with the real-coded edit distance as fitness function is the most efficient solver in terms of execution time.

5.3 RQ3: How does the proposed technique perform on large and complex industrial systems?

To address **RQ3**, we carried out experiments on two industrial systems (recall Section 4), provided by one of our industrial partners. As the experiments had to be run on a dedicated machine, only 4 TOs, one solver (the best from the previous experiments) and 3 repetitions were carried out. All of these experiments were run *after* all the other experiments were completed. In other words, all the tuning and code optimizations, that were done while experimenting with the other SUTs, were carried out *before* running the experiments on the industrial systems. Table 19 shows the results of these experiments.

In both cases, it was possible to solve at least one TO. The others are unfeasible, due to the type of input sanitization carried out by those systems. Note that whether a TO is feasible or not depends on the actual implementation of the SUT. We used the actual systems without modifications, i.e., we did not inject any artificial security vulnerability to check if our technique could spot them.

In the case of R, there was no direct mapping from the JSON fields and the fields in the XML of the TO (e.g., two of the JSON fields are concatenated in one single field in the output XML of the TO), making the search more difficult compared to M. Regarding M, interestingly, one of the fields that leads to the XML injection does get sanitized. Given the TO target field:

```
0 or 1=1
```

one of the valid inputs to solve that TO was

```
0 or 1=1<v2
```

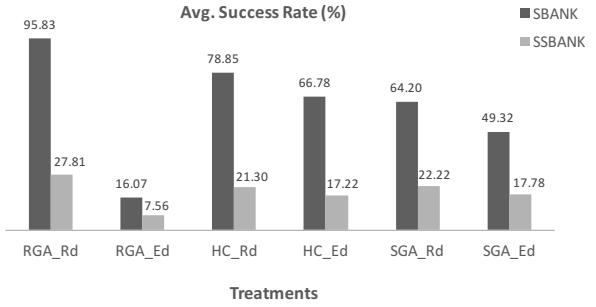


Fig. 7. Comparison of the average success rates for SBANK (without input validation) and SSBANK (with input validation).

as any character including and after the first $<$ is removed as part of the input sanitization.

Our proposed technique was able to produce inputs that can detect XMLi vulnerabilities in the evaluated industrial systems.

6 Additional Analysis

In this section, we investigate the various co-factors that may affect the effectiveness of the solvers. For this purpose, we use the two-way permutation test [48], which is a non-parametric test to verify whether such co-factors statistically affect or not the search effectiveness. This test is equivalent to the two-way Analysis of Variance (ANOVA) test [34].

Input validation: To investigate the effect of input validation, we compare the average SR for SBANK and SSBANK, which are two different front-ends for the same real-world bank card processing system. The difference is that one front-end uses input validation (i.e., SSBANK) while the other not (i.e., SBANK). This analysis can be performed by comparing the results reported in Tables 3 and 5. For each treatment, the average SR for SBANK is always higher than the SR scores achieved for SSBANK. The p-value $<$0.05 obtained from the two-way permutation test shows that the co-factor *input validation* significantly affects the performance of the solvers. This can also be observed from the bar chart depicted in Figure 7: for all the solvers, the average SR of the SBANK is always higher compared to that of SSBANK. We note that the best treatment in our study, which is RGA with Rd, could reach a success rate greater than 20% in the presence of input validation. Though there exist input validation routines in SSBANK, they are applied only on one input parameter instead of all three. Thus, if a front-end web application uses incomplete input validation, our proposed search-based technique is able to detect XMLi vulnerabilities in a reasonable amount of time (i.e., less than 10 minutes on average).

Number of input parameters: As described in Section 4.5, we have three different versions of SBANK and SSBANK with varying numbers of input parameters. This allows us to analyze how the success rate is impacted when increasing the number of input parameters. The results of this analysis are reported in Table 20, with the average SR for all the treatments with the same number of input parameters and for each application. We can clearly see that for most of the treatments the larger the number of input parameters, the smaller the success rates achieved by the different treatments. For example, for HC with Ed the success rate is 88.33% with one input parameter and it dramatically decreases to 72.50% and 39.50% with two and three input parameters, respectively. This overall pattern is also observable from the bar chart in Figure 8. The only exception to this general rule is the combination of RGA with Rd for which we can observe limited variation in the average success rate as depicted in Figure 8. Therefore, our best configuration (i.e., RGA with Rd) is little affected by increasing the number of input parameters, as opposed to the other treatments.

The permutation test for the number of input parameters also reveals a significant interaction between this co-factor and the SR (p-value$<$0.05). Hence, we conclude that increasing the number of inputs adversely affects the average SR of the solvers, i.e., the higher the number of input parameters, the more difficult is to detect XMLi vulnerabilities.

Initial population: As described in Section 4.3, the initial set of random tests can be composed by input strings with Fixed (F) or Variable (V) length. To investigate the effect of this co-factor, we compare the average SR obtained by each

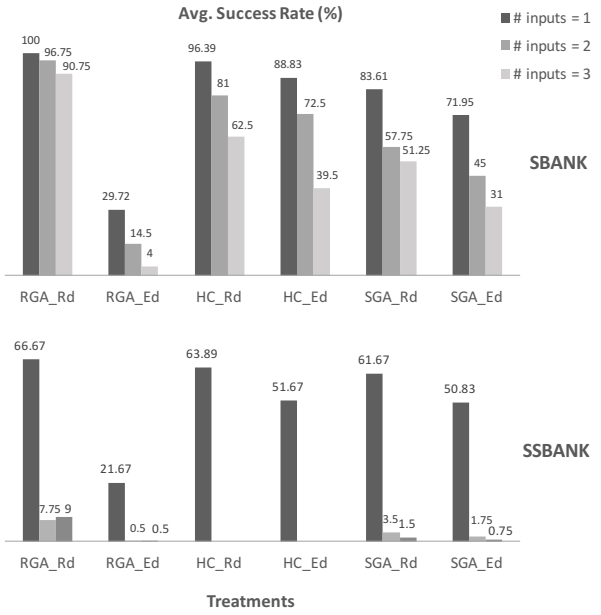


Fig. 8. Comparison of the average success rates for SBANK and SSBANK with 1, 2 and 3 input parameters.

TABLE 20
Comparison of the average Success Rates (SR) of the experiments involving applications with 1, 2 and 3 inputs

App./Solver	SBANK			SSBANK		
	1 input	2 inputs	3 inputs	1 input	2 inputs	3 inputs
RGA with Rd	100.00	96.75	90.75	66.67	7.75	9.00
RGA with Ed	29.72	14.50	4.00	21.67	0.50	0.50
SGA with Rd	83.61	57.75	51.25	61.67	3.50	1.50
SGA with Ed	71.94	45.00	31.00	50.83	1.75	0.75
HC with Rd	96.39	81.00	62.50	63.89	0.00	0.00
HC with Ed	88.33	72.50	39.50	51.67	0.00	0.00
Avg/app	78.33	61.25	46.50	52.73	2.25	1.96

TABLE 21
Comparison of the average success rates (SR) when using an initial population composed by input strings with Fixed (Fix) or Variable (Var) length

Solver	SBANK		SSBANK		XMLMAO	
	Fix	Var	Fix	Var	Fix	Var
RGA with Rd	94.67	97.00	25.39	30.22	100.00	100.00
RGA with Ed	16.13	16.02	6.85	8.26	39.58	40.83
SGA with Rd	55.44	72.96	20.00	24.44	74.58	90.83
SGA with Ed	46.35	52.28	17.04	18.52	70.83	82.08
HC with Rd	81.35	78.57	20.93	21.67	97.50	95.00
HC with Ed	73.50	60.06	19.07	15.37	91.67	89.17
Avg/app	61.24	62.82	18.21	19.75	79.03	82.99

solver, for each application, when using Fixed and Variable length. The result of this analysis is reported in Table 21, which shows the average SR achieved for each solver and application. We can see that the difference between the two types of setting is limited, i.e., it is on average 1.58% for SBANK, 1.54% for SSBANK, and 3.96% for XMLMao. These small differences can be visualized through the bar chart in Figure 9 and a permutation test further shows they are not significant (p-value=0.80). Therefore, we conclude that the length of the input strings in the initial population (or the initial solution for HC) does not significantly affect the performance of the solvers.

Alphabet size: Instead of using the complete alphabet (i.e., all possible ASCII characters), we can restrict its size by considering only the characters we determine to be used in the TOs. However, as discussed in Section 4.3, this strategy may be detrimental when there is no straightforward relationship match between input strings and the generated XML messages, due to transformations and validation. Therefore, we want to analyze the impact of this strategy on the performance of the various treatments. The bar chart in Figure 10 indicates that the effect of this co-factor on the average success rate SR is very small. Only for RGA with Ed and SGA (either with Ed or Rd) we can observe slightly higher success rates when using the restricted alphabet size. The permutation test also reveals no significant interaction (p-value=0.55) between the success rate (i.e., effectiveness) and the size of the alphabet. With respect to efficiency, Figure 11 depicts the effect of the size of the alphabet on average execution time. As we can observe, the efficiency of RGA with Rd is not affected by this co-factor while all the other solvers achieved a lower execution rate with a restricted alphabet. However, the effect is still not significant according to permutation tests, i.e., p-value=0.20. Therefore, reducing the size of the alphabet is not recommended given its low impact on both effectiveness and efficiency of the various treatments (and RGA with Rd in particular) combined with the high risk of unintentionally excluding characters that may lead to *XMLi* attacks.

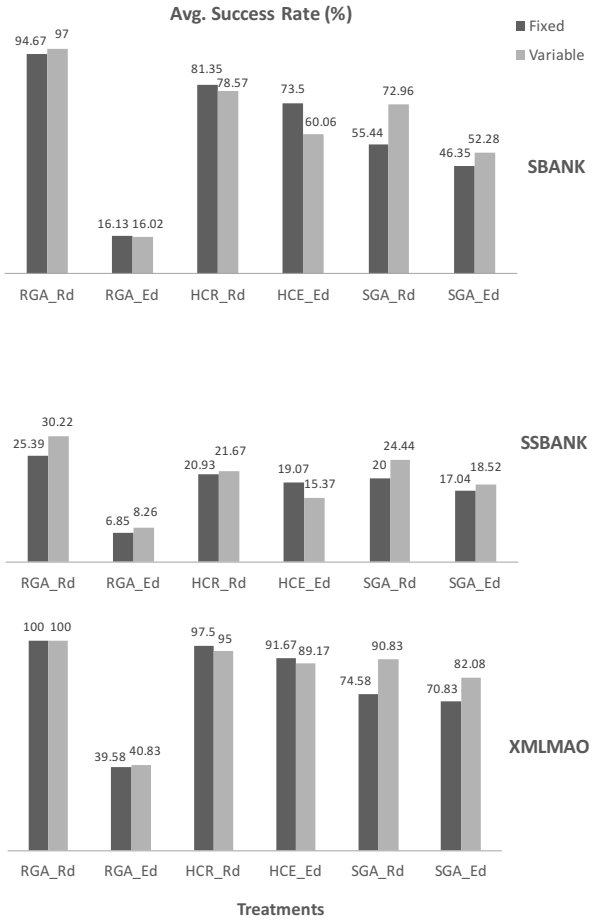


Fig. 9. Comparison of the average success rates (SR) when using an initial population composed by input strings with Fixed (Fix) or Variable (Var) length.

7 Related Work

In this section, we survey work related to vulnerability detection in web applications/services, with particular attention to XML vulnerabilities. We also discuss existing work that uses search-based approaches for security testing, including our previous work [11], which we extended in this paper.

Automated approaches for vulnerability testing: There is a large research body investigating automated approaches for the detection of vulnerabilities in web applications/services, e.g., [49], [50], [51], [52], [53]. Bau et al. [54] performed a study to evaluate the effectiveness of the state-of-the-art in automated vulnerability testing of web applications. Their results demonstrate that such approaches are only good at detecting straightforward, historical vulnerabilities but fail to generate test data to reveal advanced forms of vulnerabilities. Mainka et al. [49] presented an automated penetration testing approach and evaluated it on several web service frameworks. They implemented a tool named WSAttacker and targeted two web service specific attacks: WS-Addressing spoofing[5] and SOAPAction[6]. Their work was further extended by Oliveira et al. [51] with another tool (WSFAgresser) targeting specific web service attacks. A common issue with most of these automated approaches is the large number of false positives, which makes their application in practice difficult. Besides, none of these approaches are dedicated towards the detection of XML injections, the

5. http://www.ws-attacks.org/index.php/WS-Addressing_spoofing
6. http://ws-attacks.org/index.php/SOAPAction_spoofing

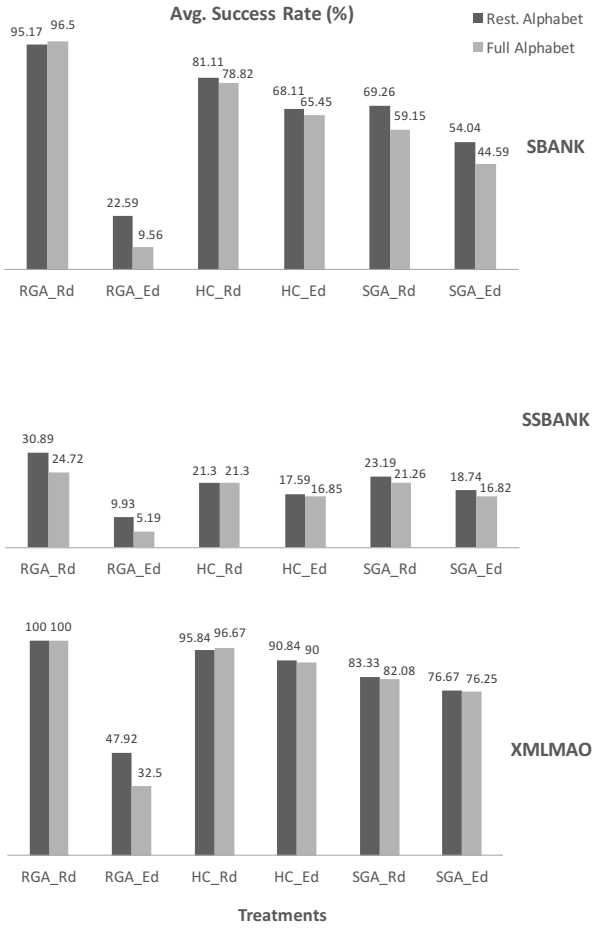


Fig. 10. Comparison of the average success rates (SR) with complete (Full) and restricted (Restricted) alphabet size.

Fig. 11. Comparison of the average execution time with complete (Full) and restricted (Restricted) alphabet size.

objective of this paper.

Testing for XML Injections: Unlike SQL injection and cross-site scripting vulnerabilities that received much attention (e.g., [55], [56], [57]), only limited research targets XML injections. An approach for the detection of XML injection attacks is presented by Rosa et al. [58]. They proposed a strategy to first build a knowledge database from the known attack patterns and then use it for detecting XML injection attacks, when they occur. This approach is an improvement over the traditional signature-based detection approaches, however it focuses on intrusion detection, not on security testing. In contrast, our work is targeted towards test data generation to detect XML injection vulnerabilities in web applications.

A basic testing methodology for XML injections is defined by OWASP [59]. It suggests to first discover the structure of the XML by inserting meta-characters in the SUT. The revealed information, if any, combined with XML data/tags can then be used to manipulate the structure or business logic of the application or web service. OWASP also provided a tool named WSFUZZER [9] for SOAP penetration testing with fuzzing features. However, as reported in [10], the tool could not be used with WSDLs having complex structure (nested XML elements) and is only useful in scenarios where the web services are directly accessible for testing.

In our previous work [10], we discussed four types of XML injection attacks and proposed a novel approach for testing web services against these attacks. Our evaluation found the approach very effective compared to state-of-the-art tools. However, it focuses on the back-end web services that consume XML messages and are directly accessible for testing. In contrast, our current work targets the front-ends (web applications) of SOA systems that produce XML messages for web services or other back-end systems.

In addition, while in [10] we used constraint solving and input mutation for manipulating XML messages, in this paper we use search-based testing techniques to generate test inputs for the front-end of the SUT that produces malicious XML messages. Such inputs can then help detect XMLi vulnerabilities in web applications that can be exploited through the front-ends.

Search-based approaches for security testing: Search-based testing has been widely investigated in literature in the context of functional testing [60], [61], [62], [63], [64]. However, little attention has been devoted to non-functional properties of the SUT, such as security testing [65], [66].

Avancini and Ceccato [67] used search-based testing for cross-site scripting vulnerabilities in web applications. Their approach uses static analysis to look for potential cross-site scripting vulnerabilities in PHP code. Then, genetic algorithms and constraint solvers are used to search for input

values that can trigger the vulnerabilities. This approach is white-box and targets a different type of vulnerabilities, i.e., cross-site scripting. Instead, our approach is completely black-box, i.e., it does not require the source code and it targets XML injection vulnerabilities.

Thomé et al. [68] proposed a search-based testing approach to detect SQL injection vulnerabilities in web applications. Their approach evolves inputs by assessing the effects on SQL interactions between the web server and database with the goal of exposing SQL injection vulnerabilities. Our work is also based on evolving test inputs but for XML injection instead of SQL. Moreover, Thomé et al. [68] used a fitness function based on a number of factors to measure the likelihood of the *SQLi* attacks. Instead, we use a fitness function based on the distance between the SUT's outputs and test objectives based on attack patterns.

Evolutionary algorithms have been also used to detect other types of vulnerabilities [69], [70]. Unlike our black-box approach for *XMLi* testing, these techniques are white-box and are focused on buffer overflow detection.

Previous work and current extension: In our previous work [11], we presented a search-based approach for generating test inputs exploiting XML injection vulnerabilities in front-end web applications. We used the standard Genetic Algorithm along with the string-edit distance to find malicious test inputs. We evaluated our approach on several web applications including a large industrial application and we also compared it with random search. We found our proposed search-based testing approach to be very effective, as it was able to cover vulnerabilities in all case studies while the random search could not, in any single case.

The current paper extends our previous work in several ways. First, we introduced a different fitness function, i.e., the Real-coded Edit Distance (Rd), which further improves the traditional string edit distance (Ed). Second, we investigated two further optimization algorithms, namely Real-coded Genetic Algorithm (RGA) and Hill Climbing (HC), in addition to the standard Genetic Algorithm (SGA) and random search (RS). Third, we enlarged our empirical evaluation using an additional industrial application. Last, we conducted an extensive evaluation by comparing all possible combinations of solvers (i.e., SGA, RGA, HC and RS) and fitness functions (i.e., Rd and Ed). Our new results show that RGA with Rd is significantly superior the previous approach [11] in terms of both effectiveness and efficiency.

8 Threats to Validity

In this section, we discuss the threats that could potentially affect the validity of our findings.

Internal validity: In our context, there are three main threats related to internal validity: (i) the use of randomized algorithms (ii) the choice of parameter settings for the algorithm and (iii) using execution time to measure efficiency of different algorithms and fitness functions.

To mitigate the first threat, we repeated each experiment several times, i.e., 10 times for each subject of Study 1 and three times for the industrial systems in Study 2, and reported the aggregated results. The use of rigorous statistical analysis also adds support to our findings [71].

To mitigate the threats arising from the parameter settings, especially for Genetic Algorithms, we used the parameter values that are recommended in the literature and also carried out some preliminary experiments before using them for the complete experiments (as described in Section 4.5). We also used the same parameter settings for all solvers. Different parameter settings could have led to different results, and the best parameters for a specific algorithm might not be the best for a different one. However, at least in the context of test data generation, where tools/algorithms are configured to work on many problem instances once released to the public, default values from the literature tend to yield good results on average [44].

Another potential threat is related to the use of execution time to compare the efficiency of different treatments. To mitigate this threat, we implemented all algorithms and fitness functions in the same tool as described in Section 4.6. Moreover, we used the same search operators (i.e., selection, crossover and mutation) available in JMetal, across all experiments, which is a well-known optimization framework. Finally, it is worth noting that the empirical study carried out in this paper is based on a software tool we developed. As for any software, although it has been carefully tested, we cannot guarantee that such tool is bug-free.

External validity: Threats to external validity concern the generalization of our findings. The empirical study is based on a small set of applications. This was due to two main reasons: First, we conducted a large empirical study with different solvers and fitness functions, which required a cluster of computers running for days. Using more subjects would have not been feasible. Second, enterprise systems are usually not accessible on open-source repositories, so we were limited by what was provided by our industrial partners. Furthermore, due to technical constraints, such systems had to be run on a dedicated machine, and not a cluster of computers.

Although this presents a threat to the generalization of our results, we have made sure to evaluate our approach with different types of applications, i.e., front-end web applications interacting with the bank-card processing system, an open source application and real-world industrial application with millions of registered users. Further, we have also evaluated applications with different levels of complexity, i.e., three versions of SBANK and SSBANK with varying number of parameters and the presence of input validation routines. Also, using real industrial systems in the case study does prove that our technique can be effective on actual systems used in practice.

Another possible threat is that in our experiments we considered only problems with up to three input fields. To the best of our knowledge, we are aware of no reliable statistics about the average number of input fields in HTML forms on the web. We hence looked at the case study of our industrial partners, where such average is 2.3, which is in line with our own experience as users of web applications. This value gives us confidence that our empirical study is not too far off from most real scenarios in practice. Moreover, all TOs in our empirical study involve up to three user inputs, as discussed in Section 4.1.

Conclusion validity: Regarding the threats to conclusion validity, we have carried out the appropriate and well-

known statistical tests along with multiple repetitions of the experiments. In particular, we have used the parametric Fisher's exact test, the non-parametric Wilcoxon test, Friedman's test and the two-way permutation test to find whether the outcomes (success rate for effectiveness and average execution time for efficiency) of the treatments differ significantly. Besides, we have also used the Odds Ratio (OR) and Vargha-Delaney ($\hat{A}_{12}$) statistics to measure the effect size, i.e., the magnitude of the observed difference. Our conclusions are based on the results of these tests and statistics.

It should also be noted that being able to carry out a successful XML injection attack does not necessarily mean that the receiver of such messages (e.g., a SOAP web service) will be compromised. This depends on how the receiver is implemented (e.g., does it have adequate level of input validation/sanitization routines?). However, in practice, internal web services (not directly accessible on the internet) might not be subject to rigorous penetration testing as the user front-end, and so might be less secure.

9 Conclusion

In this paper, we have presented an effective approach for the automated security testing of web applications based on metaheuristic search, with a focus on *XMLi* vulnerabilities. Web applications often act as front-ends to the web services of SOA systems and should not be vulnerable to malicious user inputs leading to the generation of malicious XML messages targeting these services. However, because of their complexity, effectively testing for such vulnerabilities within time and resource constraints is a challenge. In such context, *XMLi* vulnerabilities are common and can lead to severe consequences, such as DoS or data breaches. Therefore, automated and effective testing to detect and fix *XMLi* vulnerabilities is of paramount importance.

Our 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 [10].

To generate effective SUT inputs, we have presented a search-based approach. Our goal is to be able to lead the system under test (SUT) into producing malicious XML messages by effectively searching the user input space (e.g., HTML forms). In this paper, we evaluated four different search algorithms, with two different fitness functions. We have evaluated and compared them on proprietary and open source systems and two industrial systems (one being a very large web application). Our results are promising as the proposed approach was able to effectively and efficiently uncover vulnerabilities in all these case studies. In particular, the Real-coded Genetic Algorithm, using a fitness function minimizing a real-coded edit distance between TOs and generated XML messages, clearly showed to be the best algorithm and appeared to be sufficiently effective, efficient, and scalable to be used in practice.

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 would only need to modify the TOs according to the corresponding types of attacks for that vulnerability. Our future work will extend the current approach to cover more vulnerabilities.

Acknowledgements

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277). We thank Cu D. Nguyen for his contributions in our previous work on which we built on.

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] J. Ransome and A. Misra, *Core Software Security: Security at the Source*. CRC Press, 2013.

[6] 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

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

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

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

[10] 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.

[11] S. Jan, C. D. Nguyen, A. Arcuri, and L. Briand, "A Search-based Testing Approach for XML Injection Vulnerabilities in Web Applications," in *Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST 2017)*, March 2017.

[12] D. Stuttard and M. Pinto, *The web application hacker's handbook: finding and exploiting security flaws*. John Wiley & Sons, 2011.

[13] J. Clarke, *SQL Injection Attacks and Defense*, 1st ed. Syngress Publishing, 2009.

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

[15] M. Alshraideh and L. Bottaci, "Search-based software test data generation for string data using program-specific search operators: Research articles," *Softw. Test. Verif. Reliab.*, vol. 16, no. 3, pp. 175–203, Sep. 2006.

[16] L. Metcalf and W. Casey, *Cybersecurity and Applied Mathematics*. Syngress, 2016.

[17] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE transactions on evolutionary computation*, vol. 1, no. 1, pp. 67–82, 1997.

[18] S. Shamshiri, J. M. Rojas, G. Fraser, and P. McMinn, "Random or genetic algorithm search for object-oriented test suite generation?" in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '15. New York, NY, USA: ACM, 2015, pp. 1367–1374.

[19] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 254–265. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568254

[20] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, Feb. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2188385.2188395

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

[22] 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.

[23] 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.

[24] F. Herrera, M. Lozano, and J. L. Verdegay, "Tackling real-coded genetic algorithms: Operators and tools for behavioural analysis," *Artificial intelligence review*, vol. 12, no. 4, pp. 265–319, 1998.

[25] A. Ghosh and S. Tsutsui, *Advances in evolutionary computing: theory and applications*. Springer Science & Business Media, 2012.

[26] S. Picek, D. Jakobovic, and M. Golub, "On the recombination operator in the real-coded genetic algorithms," in *Evolutionary Computation (CEC), 2013 IEEE Congress on*. IEEE, 2013, pp. 3103–3110.

[27] K. Deb and D. Deb, "Analysing mutation schemes for real-parameter genetic algorithms," *International Journal of Artificial Intelligence and Soft Computing*, vol. 4, no. 1, pp. 1–28, 2014.

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

[29] P. Sprent, *Fisher Exact Test*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 524–525. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04898-2_253

[30] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum Associates Publishers, 2005.

[31] W. J. Conover", *"Practical Nonparametric Statistics"*, 3rd ed. "Wiley", "1998".

[32] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[33] S. García, D. Molina, M. Lozano, and F. Herrera, "A study on the use of non-parametric tests for analyzing the evolutionary algorithms' behaviour: a case study on the cec'2005 special session on real parameter optimization," *Journal of Heuristics*, vol. 15, no. 6, p. 617, 2008. [Online]. Available: http://dx.doi.org/10.1007/s10732-008-9080-4

[34] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of eugenics*, vol. 7, no. 2, pp. 179–188, 1936.

[35] W. J. Conover and R. L. Iman, "Rank transformations as a bridge between parametric and nonparametric statistics," *The American Statistician*, vol. 35, no. 3, pp. 124–129, 1981. [Online]. Available: http://amstat.tandfonline.com/doi/abs/10.1080/00031305.1981.10479327

[36] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian journal of statistics*, pp. 65–70, 1979.

[37] K. A. De Jong, "An analysis of the behavior of a class of genetic adaptive systems." Ph.D. dissertation, Ann Arbor, MI, USA, 1975, aAI7609381.

[38] J. Grefenstette, "Optimization of control parameters for genetic algorithms," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 16, no. 1, pp. 122–128, Jan. 1986. [Online]. Available: http://dx.doi.org/10.1109/TSMC.1986.289288

[39] 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

[40] J. D. Schaffer, R. A. Caruana, L. J. Eshelman, and R. Das, "A study of control parameters affecting online performance of genetic algorithms for function optimization," in *Proceedings of the third international conference on Genetic algorithms*. Morgan Kaufmann Publishers Inc., 1989, pp. 51–60.

[41] J. E. Smith and T. C. Fogarty, "Adaptively parameterised evolutionary systems: Self adaptive recombination and mutation in a genetic algorithm," in *Parallel Problem Solving from NaturePPSN IV*. Springer, 1996, pp. 441–450.

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

[43] 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

[44] A. Arcuri and G. Fraser, "Parameter tuning or default values? an empirical investigation in search-based software engineering," *Empirical Software Engineering*, vol. 18, no. 3, pp. 594–623, 2013.

[45] A. Panichella, F. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, 2017, to appear.

[46] A. Panichella, F. M. Kifetew, and P. Tonella, "Reformulating branch coverage as a many-objective optimization problem," in *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*. IEEE, 2015, pp. 1–10.

[47] 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

[48] R. D. Baker, "Modern permutation test software," *Randomization Tests, chapter Appendix. Marcel Decker*, 1995.

[49] C. Mainka, J. Somorovsky, and J. Schwenk, "Penetration testing tool for web services security," in *Services (SERVICES), 2012 IEEE Eighth World Congress on*, June 2012, pp. 163–170.

[50] W. Chunlei, L. Li, and L. Qiang, "Automatic fuzz testing of web service vulnerability," in *Information and Communications Technologies (ICT 2014), 2014 International Conference on*, May 2014, pp. 1–6.

[51] R. Oliveira, N. Laranjeiro, and M. Vieira, "Wsfaggressor: An extensible web service framework attacking tool," in *Proceedings of the Industrial Track of the 13th ACM/IFIP/USENIX International Middleware Conference*, ser. MIDDLEWARE '12. ACM, 2012, pp. 2:1–2:6.

[52] J. Chen, Q. Li, C. Mao, D. Towey, Y. Zhan, and H. Wang, "A web services vulnerability testing approach based on combinatorial mutation and soap message mutation," *Service Oriented Computing and Applications*, vol. 8, pp. 1–13, 2014. [Online]. Available: http://link.springer.com/article/10.1007/s11761-013-0139-1

[53] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of sql injection and cross-site scripting attacks," in *2009 IEEE 31st International Conference on Software Engineering*, May 2009, pp. 199–209.

[54] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the art: Automated black-box web application vulnerability testing," in *2010 IEEE Symposium on Security and Privacy*, May 2010, pp. 332–345.

[55] 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.

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

[57] 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.

[58] 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.

[59] "Testing for XML Injection," https://www.owasp.org/index.php/Testing_for_XML_Injection_(OTG-INPVAL-008), accessed: 2016-04-26.

[60] 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.

[61] 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

[62] P. McMinn, "Search-based software test data generation: A survey," *Softw. Test. Verif. Reliab.*, vol. 14, no. 2, pp. 105–156, Jun. 2004. [Online]. Available: http://dx.doi.org/10.1002/stvr.v14:2

[63] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 226–247, March 2010.

[64] ——, "A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA '07. New York, NY, USA: ACM, 2007, pp. 73–83. [Online]. Available: http://doi.acm.org/10.1145/1273463.1273475

[65] 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

[66] 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.

[67] 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.

[68] 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

[69] 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

[70] 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.

[71] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability (STVR)*, vol. 24, no. 3, pp. 219–250, 2014.