securityandtrust.lu

Software Verification and Validation Laboratory: Black-box SQL Injection Testing: Technical Report

Dennis Appelt, Cu Duy Nguyen, and Lionel Briand

Interdisciplinary Centre for Security, Reliability and Trust

University of Luxembourg

Nadia Alshahwan

Department of Computer Science

University College London

January 28, 2014

UNIVERSITÉ DU LUXEMBOURG

Abstract

Web services are increasingly adopted in various domains, from finance and e-government to social media. As they are built on top of the web technologies, they suffer also an unprecedented amount of attacks and exploitations like the Web. Among the attacks, those that target SQL injection vulnerabilities have consistently been top-ranked for the last years. Testing to detect such vulnerabilities before making web services public is crucial. We present in this report an automated testing approach, namely μ**4SQLi**, and its underpinning set of mutation operators. μ**4SQLi** can produce effective inputs that lead to executable and harmful SQL statements. Executability is key as otherwise no injection vulnerability can be exploited. Our evaluation demonstrated that the approach outperforms contemporary known attacks in terms of vulnerability detection and the ability to get through an application firewall, which is a popular configuration in real world.

1 Introduction

The Service Oriented Architecture (SOA) paradigm has been rapidly adopted in a wide range of areas, from financial systems, business integration and e-government to social media and end-user mobile applications. This shift has been further accelerated by the rise of cloud-based systems. There are currently more than ten thousand public web services (key units in the SOA) available[1]. The number is even larger if we also consider private web services within and among institutions.

The popularity of SOA applications can be attributed to their continuous availability, interoperability, and flexibility. However, this also makes them attractive to malicious users. The number of reported web vulnerabilities is growing sharply [12]. Recent vulnerability reports found that web-based systems can receive up to 26 attacks per minute [5]. A security testing survey of publicly available web services owned by companies such as Microsoft and Google found that at least 8% of web services contained vulnerabilities [30]. These findings, together with the high pressure of deadlines, suggest the need for *automated* security testing approaches that can cope with the increasing security risks and the limited time and resources allocated for testing.

Throughout the past decade, SQL injection (SQLi) vulnerabilities have been consistently ranked by the Open Web Application Security Project (OWASP)[2] as the top security risks. SQLi attacks target database-driven systems by injecting SQL code fragments into vulnerable input parameters that are not properly checked and sanitised. These injected code fragments could change the application's behaviour if they flow into SQL statements exposing or changing the system's data.

A large body of work in the literature addresses SQLi vulnerabilities, e.g., [13, 20, 24, 25, 27, 32]. Some approaches perform vulnerability analysis, such

[1]Data reported by http://www.programmableweb.com, accessed January 11nd 2014.
[2]https://www.owasp.org

as taint analysis [32, 34], to detect input fields that are not properly sanitised. Such approaches link to specific languages and suffer from high false positive rates and the dynamic typing and variable naming of the languages [8]. These white-box approaches often require access and may need to modify source code, which might not always be feasible when companies outsource the development of their systems or acquire third-party components. Other approaches are black-box [1, 18] but bounded to known attack patterns that tend to become out-dated very quickly as the web evolves.

In this report we propose a black-box automated testing approach targeting SQLi vulnerabilities, called μ**4SQLi**. Starting from a "legal" test case, our approach applies a set of mutation operators that are specifically designed to increase the likelihood of generating successful SQLi attacks. More specifically, new attack patterns are likely to be generated by applying multiple mutation operators on the same input. Moreover, some of our mutation operators are designed to obfuscate the injected SQL code fragments to bypass security filters, such as web application firewalls (WAF), while others aim to repair SQL syntax errors that might be caused by previous mutations. As a result, our approach can generate test inputs that produce syntactically correct and executable SQL statements that can reveal SQL vulnerabilities, if they exist. By producing SQLi attacks that bypass the firewall and result in executable SQL statements we ensure to find exploitable vulnerabilities as opposed to vulnerabilities that can not be exploited, for example because a filter blocks all attacks. In addition, concrete sample attacks produced by our approach can help developers to fix the source code or the security filter's configuration. Our approach is fully automated and supported by a tool called Xavier[3].

We have evaluated our approach on some open-source systems that expose web service interfaces. Compared to a baseline approach, called **Std**, which consists of an up-to-date set of 137 known SQLi attack patterns, our approach is faster and is significantly more likely to detect vulnerabilities within a set time budget. As a result, our approach has higher chance and is faster to detect vulnerabilities when the time budget is limited. Moreover, when the subject systems are protected by a WAF, **none** of the inputs generated by **Std** that reveal vulnerabilities can get through the firewall, while our approach can still generate a good amount of inputs, getting through the firewall and revealing all-but-one known vulnerabilities.

The remainder of this report is organised as follows: Section 2 provides a background on SQLi vulnerabilities and reviews related work. Section 3 presents our proposed mutation operators and security testing approach and tool. Section 4 presents the evaluation together with a discussion of results and threats to validity. Finally, Section 5 concludes the work.

[3]Contact us for download

2 Background and Related Work

This section provides a brief background on web services and SQLi vulnerabilities and reviews previous work on SQLi testing.

2.1 Background

In systems that use databases, such as web-based systems, the SQL statements that are used to access the back-end database are usually treated by the native application code as strings. These strings are formed by concatenating different string fragments based on user choices or the application's control flow. Once the SQL statement is formed, special functions are used to send the SQL statement to the database server to be executed. For example, an SQL statement is formed as follows (a simplified example from one of our web services in the case study):

```
$sql = "Select * From hotelList where country ='";
$sql = $sql . $country;
$sql = $sql . '"';
$result = mysql_query($sql) or die(mysql_error());
```

The variable *$country* is an input provided by the user, which is concatenated with the rest of the SQL statement and then stored in the string variable *$sql*. The string is then passed to the function *mysql_query* that sends the SQL statement to the database server to be executed.

SQLi is an attack technique in which attackers inject malicious SQL code fragments into these input parameters. Such attacks are possible when input parameters are used directly in SQL statements without proper validation or sanitisation. An attacker might construct input values in a way that changes the behaviour of the resulting SQL statement enabling the attacker to perform actions on the database that were not intended by the application's developer. These actions could lead to exposure of sensitive data, insertion or alteration of data without authorisation, loss of data or even taking control of the database server.

In the previous example, if the input *$country* has the value *' or 1=1 –*, the resulting SQL statement would be:

```
Select * From hotelList where country='' or 1=1 --'
```

The first quote closes the existing quote in the statement and the double dash at the end comments out the final quote, making the resulting SQL statement syntactically valid. The clause *or 1=1* is a tautology, i.e. the condition will always evaluate to true bypassing the original condition in the *where* clause and returning all rows in the table.

To avoid such attacks, application developers use filters and sanitisation techniques to prevent malicious inputs from affecting the application's behaviour. Developers have to be careful not to block valid inputs that might resemble malicious inputs. For example, a filter that rejects inputs with single quotes would protect from the attack in the previous example. However, this filter will also reject valid inputs where single quotes are part of a name (e.g., *O'Brain*).

Web services, the basic blocks for the service-oriented architecture, provide facilities for the easy access and exchange of information across the Web. Each web service provides a set of operations that can be invoked by clients. An operation is similar to a method in traditional programming languages, it has a set of input parameters and returns a structured output. The interface and features of a web service are typically described by a publicly available *Web Services Description Language* (WSDL) file.

In this report we consider SQLi vulnerabilities of the input parameters of a service under test: an input parameter is vulnerable to SQLi attacks if it is used in any SQL statement of the implementation of a service and if, through this parameter, an attacker can send malicious inputs that can change the intended logic of the SQL statement. To exploit such vulnerabilities, the attacker has to provide inputs that result in executable SQL statements. Otherwise, the resulting statements are rejected by the database, thus no access or changes to the data are possible.

2.2 Related Work

Previous research on SQLi detection used both white-box and black-box approaches to detect vulnerabilities. Several white-box approaches used taint analysis to identify invalidated inputs that flow into SQL statements [20, 25, 27, 32]. Fu and Qian [13] suggested using symbolic execution to identify the constraints that need to be satisfied to lead to an SQLi attack. Shar et al. [24] used data mining of the source code to predict vulnerabilities. As well as requiring access to the source code, which as we mentioned before might not always be possible, most of these approaches rely, in some aspects of their algorithms, on a set of known vulnerability patterns.

Existing black-box approaches also rely on known injection patterns when generating test cases. Ciampa et al. [6] proposed an approach that analyses the output, including error messages, of both legal and malicious test cases to learn more about the type and structure of the back-end database. This information is then used to craft attack inputs that are more likely to be successful at revealing vulnerabilities. Antunes et al. [1, 2] also analysed the difference in the behaviour of an application when using malicious and legal inputs to detect vulnerabilities. Huang et al. [18] used a test generation approach that uses known attack patterns.

Known SQLi patterns have been enumerated and discussed by various academic [1, 2, 14] and online security sources [29, 28]. However, relying on these patterns might not be sufficient to test an application as attackers are always finding new techniques to exploit vulnerabilities. Moreover, there might be a large number of different representations for the same pattern, for example, using different encodings.

Some approaches proposed run-time prevention techniques rather than testing techniques. In the majority of these approaches [15, 16, 21, 26, 33], static analysis is used to collect all possible forms of SQL statements that can be produced by the program. At run-time, if the structure of an SQL statement does not match any of those collected forms, the statement is flagged as a potential attack. Sekar [22] combined taint analysis and policies to detect injection attacks at run-time. Run-time prevention approaches are complementary to testing approaches and can also be used as an effective oracle for testing [3].

In our previous paper [3], we found that using run-time prevention techniques as an oracle improved the detection rates for SQLi testing. We also identified the need for a more sophisticated oracle that could reason about the exploitability of a discovered vulnerability. In this work, we try to address this issue by enhancing the oracle to evaluate the executability of a formed attack. A malicious input can successfully evade all security mechanisms but the resulting attack could produce an SQL statement that is not executable and, therefore, provides no evidence that the vulnerability is exploitable.

Mutation testing has been proposed and studied extensively [19] as a method of evaluating the adequacy of test suites where the program under test is mutated to simulate faults. Shahriar and Zulkernine [23] defined SQLi specific mutation operators to evaluate the effectiveness of a test suite in finding SQLi vulnerabilities. Mutation analysis was also used by Fonseca et al. [11] to compare the effectiveness of commercial security testing tools. The mutation operators we propose in this report mutate test inputs to increase the likelihood of triggering vulnerabilities rather than the program under test to evaluate the effectiveness of test suites in finding faults.

Holler et al. [17] proposed an approach to test interpreters for security vulnerabilities, such as memory safety issues, by mutating the input code. However, the difference in the context and goal requires different mutation operators and test generation techniques.

In this report, we propose a set of mutation operators that mutate inputs to possibly form new unknown SQLi attacks for web services and address limitations of previous approaches.

3 Approach

We propose an automated technique, namely μ**4SQLi**, for detecting SQLi vulnerabilities. Our technique rests on a set of mutation operators that manipulate inputs (legitimate ones) to create new test inputs to trigger SQLi attacks. Moreover, these operators can be combined in different ways and multiple operators can be applied to the same input. This makes it possible to generate inputs that contain new attack patterns, thus increasing the likelihood of detecting vulnerabilities.

Specifically, we want to generate test inputs that can bypass web application firewalls and result in executable SQL statements. A WAF may block SQLi attacks and prevent a vulnerable web service from being exploited. Therefore, effective test inputs need to get through the WAF in order to reach the service. Furthermore, they should lead to executable SQL statements as otherwise, security problems are unlikely to arise since the database engine would reject them. And consequently, no data would be leaked or compromised.

This section introduces our proposed mutation operators to generate test data. For each mutation operator, along with its definition, a concrete example is provided. In some operators we discuss also their preconditions with respect to input and previously applied operators. We will then discuss our test generation technique and the automated tool we developed to support the technique.

3.1 Mutation Operators

Mutation operators (MO) can be classified by their purpose into the following three classes: *Behaviour-changing*, *syntax-repairing* and *obfuscation*. Table 1 provides a summary of all mutation operators.

Table 1: Summary of mutation operators classified into behaviour-changing, syntax-repairing, and obfuscation operators.

MO name	**Description**
	Behaviour-Changing Operators
MO_or	Adds an OR-clause to the input
MO_and	Adds an AND-clause to the input
MO_semi	Adds a semicolon followed by an additional SQL statement
	Syntax-Repairing Operators
MO_par	Appends a parenthesis to a valid input
MO_cmt	Adds a comment command (-- or #) to an input
MO_qot	Adds a single or double quote to an input
	Obfuscation Operators
MO_wsp	Changes the encoding of whitespaces
MO_chr	Changes the encoding of a character literal enclosed in quotes
MO_html	Changes the encoding of an input to HTML entity encoding
MO_per	Changes the encoding of an input to percentage encoding
MO_bool	Rewrites a boolean expression while preserving it's truth value
MO_keyw	Obfuscates SQL keywords by randomising the capitalisation and inserting comments

3.1.1 Behaviour-changing

This class of mutation operators mutates inputs with the aim of changing the application's expected behaviour if the application is vulnerable to SQLi. For example, the mutated input could cause the application to return more database rows than expected, exposing sensitive data to an unauthorized user. We define the following behaviour-changing operators:

Operator: MO_or

Adds *OR x=x* to the *WHERE* clause of a SQL statement where x is a random number or a character enclosed in single or double quotes.

Rationale:

The conditions in a *WHERE* clause are used to limit the rows affected by the SQL statement. By adding a condition that always evaluates to true (tautology) to the *WHERE* clause, all rows in the relevant table(s) will be affected, overriding any row filtering logic that was intended by the developer of the application. This will return all data in the related tables in case of a *SELECT* statement or delete/update all rows in case of a *DELETE* or *UPDATE* statement.

Example: from original input: *1*; **MO_or** produces a mutated input: *1 OR 1=1*. As a result, if the SQL statement that takes the input is predefined as *"SELECT * FROM table WHERE id=" + input*, the input will change the logic of the statement and turns it as follows: *SELECT * FROM table WHERE id=1 OR 1=1*. This resulting statement will return all the data of *table*.

Operator: MO_and

Adds *AND x=y* to the *WHERE* clause of a SQL statement where x and y are random numbers or single characters enclosed in single or double quotes and x is not equal to y.

Rationale:

By adding a contradiction to the *WHERE* clause, no rows will be affected by the SQL statement. This type of malicious input cannot be used to exploit a vulnerability but because the result of such input is known, this type of input can be used to confirm that a vulnerability is present.

Preconditions:

MO_or has not been applied.

Example: original input: *1*, mutated input: *1 AND 1=2*. That will turn, for example, a predefined statement: *"SELECT * FROM table WHERE id=" + input* to: *SELECT * FROM table WHERE id=1 AND 1=2*, thus, changing the logic of the original statement.

Operator: MO_semi

Adds a semicolon (;) followed by an additional SQL statement to the input. The resulting query has the form *sql_stmt1; sql_stmt2*, where *sql_stmt1* is the original SQL statement and *sql_stmt2* is an arbitrary SQL statement.

Rationale:

SQL statements separated by a semicolon are executed by the server from left to right, unless an error is encountered. If this operator is applied successfully, any SQL statement can be injected after the semicolon giving the attacker complete control over the database if no other restrictions are applied (e.g., the database user has no privilege restrictions defined on the database level).

Example: original input: *1*, mutated input: *1; SELECT waitfor(5) FROM dual.* This changes the predefined statement: *"SELECT * FROM users WHERE id=" + input* to: *SELECT * FROM users WHERE id=1; SELECT waitfor(5) FROM dual.*

3.1.2 Syntax-repairing

As mentioned before, an SQLi attack aims to change the behaviour of the application by injecting malicious inputs. Therefore, the malicious input itself is expected to contain SQL statement fragments. This type of input, unlike regular valid inputs, could cause a SQL syntax error when being combined with its targets, i.e., predefined SQL statements. Since the approach we propose is a black-box technique, the predefined SQL statement syntax is unknown to the test generator making it challenging to generate inputs that do not cause syntax errors. This class of mutation operators mutates inputs with the goal of repairing SQL syntax errors when they are encountered. The mutation operators we define in this class are the following:

Operator: MO_par

Appends a closing parenthesis to the end of an input.

Rationale:

In some cases, an input provided by the user is used as a parameter for a SQL function call or within a nested *SELECT* statement. In such scenarios the input is inserted within parenthesis, for example, *func_name(input)*. If such input is vulnerable to injections, this vulnerability can only be exploited when the opening parenthesis of the function call is matched with a closing parenthesis. Otherwise, the injected input would be interpreted as part of the function's parameter, which might cause a syntax error or cause the injection to have no effect on the application's behaviour.

Example: original input: *67*, mutated input: *67).* When the input is further mutated with **MO_or** and **MO_cmt**, the obtained mutated input will be: *67) OR 1=1* -{}-. Let us consider a predefined statement: *"SELECT * FROM table WHERE character=CHR(" + input + ")"*, where function *CHR* converts an integer to its corresponding Unicode character. The changed SQL statement: *SE-*

*LECT * FROM table WHERE character=CHR(67) OR 1=1 –).*

Operator: MO_cmt

Adds a SQL comment (double dashes -- and the hash character #) to the input. Any SQL that follows a comment is not executed.

Rationale:

A SQL comment can be useful to repair syntax errors that were caused by previous mutations. By appending a SQL comment at the end of the mutant, everything following the comment will be ignored by the parser, which might help fixing SQL syntax errors.

Preconditions:

Another operator, such as MO_par, has been previously applied and caused a syntax error.

Example: original input: *67*, after being mutated with **MO_or** and **MO_par**: *67) OR 1=1*. This changes the predefined statement: *"SELECT * FROM table WHERE character=CHR(" + input + ")"* to a combined statement, which causes a syntax error: *SELECT * FROM table WHERE character=CHR(67) OR 1=1)*.

We then apply **MO_cmt** to obtain: *67) OR 1=1 #*. The final statement: *SELECT * FROM table WHERE character=CHR(67) OR 1=1 #)* Applying this mutation causes the last parenthesis to be ignored by the parser, thereby avoiding parser error due to the unbalanced number of parentheses.

Operator: MO_qot

Adds either a single quote (') or a double quote (") to the mutant.

Rationale:

If an input that is vulnerable to injections is of type string, it may be enclosed in single or double quotes in the predefined SQL statement. The SQL parser will treat the mutant, including the injected SQL, as a string literal and will not execute any SQL commands within the mutant. To be able to exploit the vulnerability, we have to first exit the string context by closing any open quotes before any SQL commands can be injected.

Example: original input: Smith, mutated with **MO_or**: *Smith OR 1=1*. This changes the predefined statement: *"SELECT * FROM table WHERE name='" + input + "'"* to aombined statement, which does not result in the desired change of behavior, since the mutant is treated as a string literal: *SELECT * FROM table WHERE name='Smith OR 1=1')*. After being further mutated with **MO_qot** and **MO_cmt**: *Smith' OR 1=1 #*, the final statement is *SELECT * FROM table WHERE name='Smith' OR 1=1 #)*, which is syntactically correct and changes the logic of the original statement.

3.1.3 Obfuscation

Some applications employ input filters to defend against SQLi attacks. A filter examines every input to check for suspicious string patterns typically used in an

SQLi attacks, such as SQL keywords, and rejects any such inputs. For example, a filter uses a blacklist that defines forbidden characters or strings to decide if an input is suspicious. Obfuscation mutation operators try to avoid filtering by mutating an input to a semantically equivalent input but in a different form. This might prevent the filter from recognizing the forbidden characters/strings in the mutated input. We define the following obfuscation mutation operators:

Operator: MO_wsp

Replaces a whitespace with a semantically equivalent character (+, /**/, or unicode encodings: %20, %09, %0a, %0b, %0c, %0d and %a0).

Rationale:

An application might filter inputs that contain string patterns known to be used in SQLi attacks, for example, a single quote followed by a whitespace. Representing the whitespace in a different encoding might cause the malicious input to bypass this filter.

Preconditions:

The input contains at least one whitespace.

Example: original input: *1 OR 1=1*, mutated input: *1+OR+1=1*. This changes the predefined statement: *"SELECT * FROM table WHERE id=" + input* to *SELECT * FROM table WHERE id=1+OR+1=1*.

Operator: MO_chr

Replaces a character literal enclosed in quotes (*'c'*) with an equivalent representation, where c is an arbitrary printable ASCII character. Equivalent representations are:

- Short binary representation, for example, *'a'* is replaced with *b'1100001'*.
- Long binary representation, for example, *'a'* is replaced with *_binary'1100001'*.
- Unicode representation, for example, *'a'* is replaced with *n'a'*.
- Hexadecimal representation, for example, *'a'* is replaced with *x'61'*.

Rationale:

If a filter rejects a mutant generated by a behaviour-changing mutation operator which contains a character literal, this operator can be used to obfuscate the mutant. For example, MO_or generates mutants with a tautology, e.g. *OR 'a'='a'*. A filter may be configured to identify suspicious inputs by checking for a tautology pattern. To avoid this filter, this operator changes the representation of one of the tautology's operands, while preserving the semantic meaning. For example, *OR 'a'='a'* could be mutated to *OR 'a'=x'61'*, where *x'61'* is the hexadecimal representation of *'a'*. This new mutant might not be recognized as a tautology by the filter and, therefore, avoid filtering.

Preconditions:

A behaviour-changing mutation operator, which contains a character literal, has been previously applied.

Example: original input: 1, mutated with **MO_or**: *1 OR 'a'='a'*, further mutated with **MO_chr**: *1 OR 'a'=x'61'*. This changes the predefined statement: *"SELECT * FROM table WHERE id=" + input* to: *SELECT * FROM table WHERE id=1 OR 'a'=x'61'*.

Operator: MO_html

Changes the encoding of a mutant using HTML entity encoding. In HTML entity encoding, a character can be encoded in two ways: (i) numeric character reference in the form *&#N* where *N* is the character's code position in the used character set in decimal or hexadecimal representation; (ii) Character entity reference [31] in the form *&SymbolicName*. For example, *&quot;* is the encoding for the single quote character (').

Rationale:

Using HTML entity encoding might help evade a filter that is designed to reject a certain character, but does not recognize the same character if received in HTML entity encoding.

Preconditions:

For character entity reference encoding, only characters with symbolic names can be encoded.

Example: original input: 1, mutated with **MO_or**: 1 OR 'a'='a', further mutated with **MO_html**: *1 OR &quot;a&quot; = &quot;a&quot;*. This turns the predefined statement: *"SELECT * FROM table WHERE id=" + input* to: *SELECT * FROM table WHERE id=1 OR &quot;a&quot; = &quot;a&quot;*.

Operator: MO_per

Changes the encoding of a mutant using percent encoding:*%HH*, where *HH* is a two digit hexadecimal value referring to the character's ASCII code. For example, the single quote character (') is encoded as *%27*.

Rationale:

Using percent encoding is useful if a filter rejects a certain character, but does not recognize the same character if received in percent encoding.

Example: original input: 1, mutated with **MO_or**: *1 OR 'a'='a'*, further mutated with **MO_per**: *1 OR%20'a'='a'*. This turns the predefined statement: *"SELECT * FROM table WHERE id=" + input* to *SELECT * FROM table WHERE id=1*
OR%20'a'='a'.

Operator: MO_bool

Replaces a boolean expression with an equivalent boolean expression. For example, the boolean expression *1=1* which is used in MO_or could be obfuscated as *not false=!!1*. Both expressions evaluate to true, which maintains the same semantic meaning of the mutant after obfuscation.

Rationale:

A filter might be configured to look for and reject certain boolean expressions which are used as part of a request frequently used in SQLi attacks, e.g., a tautology. By obfuscating the boolean expression, the filter might fail to recognise the attack, making it possible to perform it.

Preconditions:

Can only be applied to input values that contain a boolean expression.

Example: original input: *1*, mutated with **MO_or**: *1 OR 1=1*, further mutated with **MO_bool**: *1 OR not false=!!1*. This turns the predefined statement *"SELECT * FROM table WHERE id=" + input* to: *SELECT * FROM table WHERE id=1 OR not false=!!1*.

Operator: MO_keyw

Obfuscates SQL keywords and operators using different techniques: Randomly changing the case of some letters, adding comments in the middle of a keyword or replacing a keyword with an alternative representation. Most SQL parsers are case insensitive, e.g. the keyword *select*, *SELECT* or *SeLeCt* are all valid. Some parsers accept keywords which contain a comment in the middle of the keyword (e.g. *sel/*comment here*/ect*). Finally, some keywords have alternative forms, e.g. *OR* can also be expressed as ——.

Rationale:

A filter might be configured to reject a request that contains any SQL keyword, since SQL keywords are frequently used in SQLi attack strings. By obfuscating the SQL keywords in an input, the filter may fail to recognise those keywords, making the attack successful.

Preconditions:

The input value contains at least one SQL keyword.

Example: original input: *1*, mutated with **MO_or**: *1 OR 1=1*, further mutation with **MO_keyw**: *1 —— 1=1*. This changes the predefined statement: *"SELECT * FROM table WHERE id=" + input* to: *SELECT * FROM table WHERE id=1 —— 1=1*.

3.2 Test Generation

Multiple mutation operators from different classes can be applied to a single input parameter to generate desired input vectors. The aim is to detect subtle vulnerabilities that can only be triggered with an input generated by combining multiple mutation operators. For example, consider an application that filters inputs by searching for known attack patterns that can be generated using one of the behaviour-changing operators. To form a successful attack, it is necessary to first apply the behaviour-changing operator and then apply one or more obfuscation operators.

Each chain of mutations has to start from a valid test case, which satisfies the input validations of the application under test. Starting from a valid test case ensures that we avoid generating test cases that would be directly rejected by the application due to dependencies between inputs or complex input structures that are unlikely to be generated randomly. Valid test cases have the benefit of being more likely to satisfy input validations and reach critical parts of the application, such as SQL queries. For example, if an application expects a credit card number together with other inputs, which we wish to mutate, the credit card number has to follow a well-defined format; otherwise the test case would be instantly rejected.

Algorithm 1 formally defines the test generation algorithm: Starting from a valid test case, each input is mutated a predefined number of times. The function *Apply_MO* (Line 4) randomly applies one or more mutation operator(s) to the current *Input*. The function uses a simple grammar that defines the different legal ways to combine operators. The operation under test is then called with the updated test case *TC'*. If the oracle flags a vulnerability, all SQL statements that were issued as a result of the call are checked. If the percentage of executable SQL statements (i.e., statements that do not contain a syntax error) is above a predefined threshold, the input is reported as vulnerable and the test case is saved to help the test engineer in debugging and fixing the vulnerability (Line 5-8).

Algorithm 1 Test Generation Algorithm:

```
Input  $TC$: A test case: ArrayOf($Input$)
    $OP$: A web service operation to be tested
Output  $TS$: Test Suite for SQLi vulnerabilities
    $V$: Set of vulnerable inputs
 1: $TS = \emptyset$
 2: for each $Input$ in $TC$ do
 3:   while max_tries_not_reached do
 4:     $TC' = $ apply_MO($TC$,$Input$)
 5:     if call($OP$,$TC'$) $=$ $VulnrFlagged$ then
 6:       if executable_SQL $\geq$ P then
 7:         $V = V \cup Input$
 8:         $TS = TS \cup TC'$
 9:       end if
10:     end if
11:   end while
12: end for
13: return  $TS$, $V$
```

Figure 1 shows an example of a SOAP message (a test case) generated by our approach. Here the input values of the parameters *minPrice*, *maxPrice*, and *start* are kept from the original test case, while the input value of the parameter *country* has been mutated to contain a SQLi attack.

```
<soapenv:Envelope>
  <soapenv:Header/>
  <soapenv:Body>
    <urn:getRoomsByRate>
       <minPrice xsi:type="xsd:float">100</minPrice>
       <maxPrice xsi:type="xsd:float">400</maxPrice>
       <country xsi:type="xsd:string">"||not 0--</country>
       <start xsi:type="xsd:integer">1</start>
    </urn:getRoomsByRate>
  </soapenv:Body>
</soapenv:Envelope>
```

Figure 1: Example of a generated test case, the parameter *country* contains a mutated SQLi attack.

To decide if a vulnerability is detected, we use two approaches: First we use a run-time prevention tool (database proxy). Database proxies reside between the database and the application and monitor every issued statement for SQLi attacks. Once a vulnerability is flagged, we examine all SQL statements that were issued on that specific call to check how many statements are executable. This is to check if the generated inputs caused syntax errors and, therefore, do not provide evidence that a vulnerability is exploitable, as discussed before.

3.3 Test Oracle

When a malicious input is sent to a target system, it may result in making the system misbehave if successful. In most cases, the manifestation of abnormal behaviours can be observed though the results the target system returns (e.g., web pages showing unintended content) or through the surrounding environment (e.g., crashes, illegal calls to the operating system, or unintended accesses to data). In our experiments, because of our focus on SQL injections, we deploy a database proxy that intercepts the communication between the target system and its database, to identify if an input is potentially harmful or not. For example, we can use **GreenSQL** [4] for this purpose. A previous study that compared GreenSQL to five similar tools has found it to be the most effective in detecting SQL injection attacks [10].

Details of using a database proxy as oracle has been discussed in our previous work [3]. Typically, a database proxy is deployed and trained with normal database accesses. Such training data are the results of regular usage of the systems or the execution of existing functional test suites. Based on the training data, the proxy learns regular patterns of legal SQL statements. Once trained, the proxy will continue observing the traffic between the system and its database and raise alarms when identifying suspicious database queries. Each alarm corre-

[4]http://www.greensql.com

sponds to one database SQL statement, and one test case can result in multiple SQL statements and thus multiple alarms. To avoid false positives due to incomplete training, manual inspection may be needed to verify that all SQL statements flagged did actually point to a vulnerability in the system and were not legal statements which had simply not been learned.

3.4 Tool

The presented mutation approach has been implemented as a Java tool, called Xavier[5]. It can be used to test SOAP-based web services for SQLi vulnerabilities. Figure 2 shows the key components of the tool (*Test generator* and *Monitor*) and how it is used in practice. The test generator takes as inputs the WSDL file of the web service under test and a sample test case for each web service operation that has to be tested. Such a sample test case can be easily generated by professional tools, such as SoapUI[6], or by existing approaches [4]. The tool, then, examines the sample test case to find all input parameters for an operation and replaces each parameter, one at a time, with an SQLi attack generated with our mutation approach. The modified test case will be sent to the web service under test (the SUT in the figure). In some settings, there could be a web application firewall (the WAF component) deployed in between the test generator and the SUT. The goal of the WAF is to protect the SUT from malicious attacks. The oracle component (the *DB proxy* component in the figure) observes the interactions between the SUT and its database to detect malicious SQL statements. Finally, the *Monitor* component of Xavier constantly queries the oracle component to know whether generated inputs reveal a SQLi vulnerability.

In Xavier, we integrate, as DB proxy oracle, a database security suite, called GreenSQL[7], that intercepts the SQL statements sent between the web service application and its database. The database proxy uses a learning approach to detect SQLi vulnerabilities. Therefore, the proxy has to be trained in a learning phase to recognise legal SQL statements. In the detection phase, the proxy considers all intercepted statements, which have not been learned previously, as SQLi attacks. Thus, a test case is said to reveal a vulnerability if the intercepted SQL statements are malicious.

Every suspected malicious statement is further analysed if it forms syntactically correct SQL. An attacker is only able to exploit a SQLi vulnerability, if he can inject the malicious input in such a way that the resulting SQL statement is free of syntax errors. Otherwise, the attacker is unable to reach his goal, e.g., to obtain/modify data or change the application's control flow, if the malicious statement is not executed. The tool MySQL-Proxy[8] is used to monitor if a SQL statement has been executed or if there was an error during execution.

[5]Contact us for download

[6]http://www.soapui.org

[7]http://www.greensql.com

[8]http://dev.mysql.com/doc/refman/5.1/en/mysql-proxy.html

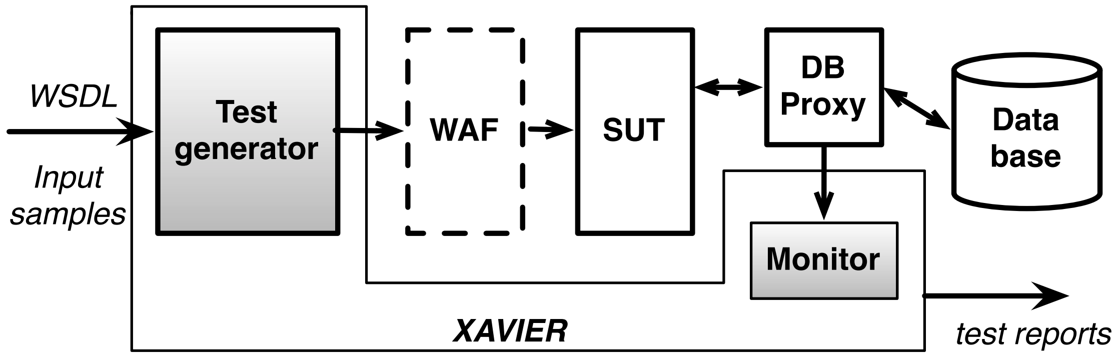

Figure 2: Components of Xavier and how Xavier is used in practice.

4 Experiments and Results

We have evaluated the effectiveness of our approach on two open source systems and in two different settings: with and without the presence of a web application firewall (WAF). The main motivation for the latter is that, in most contexts, including those of our industry partners, such a firewall is typically present (or sometimes integrated) and is the first protection layer encountered by attackers. Such situations are therefore deemed more realistic. The firewall deployed in our experiment is ModSecurity with the OWASP Core Rule Set (version 2.2.0). As the baseline for our evaluation, we considered a comprehensive list of known SQLi attack patterns since, in practice, this is what penetration testers typically use.

We aim at evaluating the performance of our proposed mutation technique in comparison with standard attacks. More specifically, we investigate the following research questions:

RQ1: *Are standard attacks and mutated inputs (generated by μ**4SQLi**) likely to reveal exploitable SQLi vulnerabilities?*

RQ2: *With and without the presence of the WAF, which input generation technique performs better?*

4.1 Subject Applications

Two open-source subjects, namely **HotelRS** and **SugarCRM** were used in our experiments. **HotelRS** was created by researchers to study service-oriented architectures and was used in previous studies [7]. **SugarCRM** is a popular customer relationship management system (received 189K+ downloads as of 2013[9]). These systems provide web service APIs to the external world. Such interfaces allow other systems, namely service consumers, to access to the business functionality and data of the subject systems. However, they are also target for SQLi attacks if the inputs through those interfaces are inadequately treated.

[9]http://sourceforge.net

Table 2: Size in terms of web service operations, parameters, and lines of code of the subject applications.

Application	#Operations	#Parameters	#LoC
HotelRS	7	21	1,566
SugarCRM	26	87	352,026
Total	33	108	353,592

Table 2 provides information about the number of operations, input parameters and lines of code for the chosen applications. In terms of size in number of lines of code, **HotelRS** and **SugarCRM**, with 1.5KLoCs and 352KLoCs respectively, are not particularly large but they provide a respectable number of services with many input parameters, with known vulnerabilities. **SugarCRM** and **HotelRS** are both implemented using PHP, use a MySQL database, and provide a SOAP-based Web Service API. Those are popular technologies used in the implementation of many web services.

4.2 Treatments

We refer to the baseline approach consisting of 137 known attack patterns as **Std** (Standard attacks). Such patterns were consolidated in a repository of SQLi attack patterns [1]. They include different contemporary categories of attacks, such as *Boolean-based*, *UNION query-based*. In the context of our study, the whole set of attack patterns of **Std** is applied for every individual input parameter. The second treatment used in our study is our approach, μ**4SQLi**.

4.3 Variables

We used **GreenSQL** [10] as the oracle (database proxy) for the generated test inputs of **Std** and μ**4SQLi**. To train GreenSQL, we have created a test suite for each of the subjects consisting of a wide range of legitimate input values. To avoid false positives due to incomplete training, in our experiments, we manually verified that all SQL statements flagged by GreenSQL did actually point to a vulnerability in the system and were not legal statements which had simply not been learned.

Given a set of test cases targeting a specific web service parameter, we define T as the total number of test cases that generate SQL statements that are flagged (alarm) by the database proxy. Among these tests, we further investigate if their generated SQL statements are executable or not. We refer to T_e for the total number of tests that can lead to flagged and executable SQL statements. To compare **Std** and μ**4SQLi**, we need to consider both T and T_e, as we will see that looking at T alone (as do many studies in the literature) would lead to very different conclusions since only executable SQL statements can be exploitable. Non-executable statements can be generated because the corresponding inputs,

[10]http://www.greensql.com

after being processed by a target, result in syntax-errors. Such statements hardly have a security impact since the database engine would reject them and, hence, no data would be leaked or compromised.

If a technique yields higher T_e, it is considered to be more effective at detecting exploitable vulnerabilities. In other words, when T_e is high, it is more likely to detect exploitable vulnerabilities for a test suite of fixed size. Moreover, it is also likely to detect vulnerabilities faster, i.e., we need a smaller number of tests to be executed in order to detect the vulnerabilities. This, in practice, is important when dealing with a large number of services and input parameters.

Since one test case can give rise to multiple SQL statements, we need to determine how to compute T_e when there is a mix of executable and non-executable statements. Since, in practice, one single flagged and executable statement generated by a specific input can entail serious consequences, when more SQL statements are executable, the chance to uncover vulnerabilities is higher. In our analysis, with the intent of being conservative in our results, a test t is considered to be part of T_e if and only if all the flagged statements generated by t are executable.

4.4 Results

We ran μ**4SQLi** and **Std** on every parameter of the two selected subjects, **SugarCRM** and **HotelRS**. There are in total 108 input parameters for all their web services. As described earlier, **Std** entails 137 test executions for every parameter, whereas with μ**4SQLi**, since it is non-deterministic, we need to run more test executions to account for randomness. To do so in an efficient way, given the substantial execution time (about 5.7 hours per vulnerable parameter on a virtual machine of 1Gb RAM and 2,6Ghz CPU) we generated and ran 1000 tests for each parameter. We, then, adopted a Bootstrapping approach (sampling with replacement) [9] and formed 10k test suites (each has 137 tests) by sampling from these 1000 tests, so that each test suite would be comparable with **Std** with respect to T_e. In the tables 3 and 4, we report the percentage of T and T_e for **Std** on each subject and their average percentage for μ**4SQLi** over 10k test suites. We only report results for vulnerable input parameters as per the results of the two test techniques and after being confirmed through manual inspection.

Table 3 shows our results when subjects were not protected by the WAF. The first and second column indicate the subjects and their vulnerable parameters, the subsequent columns show the percentage of tests that generate flagged SQL statements ($\%T$) and the percentage of such flagged tests (out of 137) that also lead to executable SQL statements ($\%T_e$). For μ**4SQLi**, as indicated, such percentages are averages over 10k test suites. For **HotelRS**, both techniques find six SQLi vulnerabilities. With regards to the parameter *country*, 12.41% of 137 test cases provided by **Std** are flagged by GreenSQL as SQLi attacks and, among them, 5.84% generate executable SQL statements. Results for the remaining five parameters found to be vulnerable by **Std** are identical: 35.04% of the tests lead to SQL statements being flagged by GreenSQL and among them, 9.49% generate executable SQL statements. While μ**4SQLi** and **Std** detect the same vulnerabilities,

Table 3: Results of **Std** and μ**4SQLi** on the subject applications when no WAF is enabled.

Subject	**Parameter**	**Std**		μ**4SQLi**	
		$\%T$	$\%T_e$	$\%T$ (avg)	$\%T_e$ (avg)
HotelRS	country	12.41	5.84	40.62	21.80
	arrDate	35.04	9.49	42.05	12.50
	depDate	35.04	9.49	42.96	12.03
	name	35.04	9.49	43.36	12.91
	address	35.04	9.49	39.81	11.00
	email	35.04	9.49	41.73	11.24
SugarCRM	value	37.23	0.0	41.48	22.51
	ass_user_id	32.85	8.03	42.49	13.91
	query1	32.85	3.65	9.82	0.30
	query2	54.74	5.84	81.72	33.45
	order_by	59.85	10.95	85.98	33.55
	rel_mod_qry	47.45	2.92	49.79	0.00

$\%T$ and $\%T_e$ are higher for μ**4SQLi**: across reported parameters, T ranges from 39.81% to 43.36% and T_e from 11% to 21.80%. For **SugarCRM**, both techniques detect five out of six vulnerabilities, but both **Std** and μ**4SQLi** failed to generate an executable SQL statement for one parameter, that is *value* and *rel_mod_query*, respectively. Except for parameter *query1* μ**4SQLi** has always a higher T measure. Similarly, μ**4SQLi** has a higher T_e measure for all parameters except for *query1* and *rel_mod_query*.

Even when using μ**4SQLi**, $\%T_e$ is generally lower than $\%T$ across input parameters. However, it is large enough to be highly likely to detect an exploitable vulnerability by running a few dozens test cases or less, as only one flagged test case leading to an executable SQL statement is enough to demonstrate the vulnerability of a parameter. Taking the parameter *ass_user_id* as an example, with a average $\%T_e$ of 13.91%, running 50 test cases would yield a very small probability, 0.0006 (i.e., $(1 - 0.1391)^{50}$), of missing the vulnerability. $\%T$ is typically much larger than $\%T_e$, thus showing that generating executable SQL statements is rather difficult.

Table 4 shows the results of the experiments when the subjects were protected by the WAF. For **HotelRS**, once again, both approaches were able to generate, for each vulnerable parameter, SQLi statements which was flagged by GreenSQL ($\%T$ ¿ 0). However, one important difference is that only μ**4SQLi** was able to generate test cases which lead to executable SQL statements. **Std** failed to do so for *all* tested parameters. Similarly, for **SugarCRM**, $\%T$ is significantly higher for μ**4SQLi** than **Std**. And once again, only μ**4SQLi** was able to generate test cases that led to executable SQL statements for five out six vulnerable parameters (except *rel_mod_qry*), whereas **Std** failed to do so for *all* tested parameters. Our conclusions are similar to the results when no WAF is present, except that $\%T$ and

Table 4: Results of **Std** and μ**4SQLi** on the subject applications protected by the WAF.

Subject	**Parameter**	**Std**		μ**4SQLi**	
		$\%T$	$\%T_e$	$\%T$ (avg)	$\%T_e$ (avg)
HotelRS	country	0.73	0.0	36.84	20.69
	arrDate	2.19	0.0	35.91	9.11
	depDate	5.84	0.0	36.59	11.42
	name	6.57	0.0	38.34	11.72
	address	7.30	0.0	39.67	9.64
	email	6.57	0.0	36.33	9.88
SugarCRM	value	2.19	0.0	37.42	20.48
	ass_user_id	5.11	0.0	29.35	6.89
	query1	0.73	0.0	8.97	0.20
	query2	3.65	0.0	76.56	31.43
	order_by	7.30	0.0	80.08	31.96
	rel_mod_qry	6.57	0.0	44.82	0.0

$\%T_e$ tend to be lower with a WAF. This is to be expected as some of the attacks generated are filtered out by the WAF.

Regarding the performance of μ**4SQLi** on the parameter *query1*, the vulnerability, though not impossible to find, is still extremely difficult to detect (only 0.3% of test cases can uncover it). Further work is needed to investigate the reasons.

We further examined why μ**4SQLi** experienced, for parameter *rel_mod_qry*, a sharp drop from T (49.72% without WAF) to T_e (0%). μ**4SQLi** failed to trigger an executable statement for this parameter since, given the SQL statement into which the test case is injected, non of the mutation operators could possibly result in a syntactically correct statement. The vulnerable statement is:

*SELECT opportunity_id id FROM accounts_opportunities , opportunities WHERE [...] AND <**test case inserted here**> AND [...]*

The injection occurs in the *where* clause of the SQL statement. **MO_or** and **MO_and** are the mutation operators that target SQLi vulnerabilities in the where clause. For both of these operators, all generated mutants for this particular SQL statement begin either with a single quote or a double quote, e.g. *"——'d'='d'–* or *' or 1*, but since their is no matching opening single or double quote an syntax error is introduced. For example, once concatenated with the mutant the statement becomes:

SELECT opportunity_id id FROM accounts_opportunities , opportunities WHERE [...] AND "——'d'='d'– AND [...]

This problem can be solved by improving how the mutation operators append a clause. For example, in this particular case, starting the mutant with a number instead of a quote prevents a syntax error. With this additional fix, the vulnerability will be detected. More generally, we expect that the performance of μ**4SQLi** will be further improved once we improve the mutation operators.

Answering the research question **RQ1**, we can see that both techniques can, in most cases, reveal vulnerabilities ($\%T_e > 0$) when the subjects were not protected by the WAF. However, when they are protected, only μ**4SQLi** can reveal such vulnerabilities (in 10 out of 12 parameters) while **Std** revealed none of them. Such a difference is highly significant as it has many practical implications to be discussed below.

***RQ1**: Both the mutation-based technique and the standard attack patterns can reveal SQLi vulnerabilities when no firewall was used. Most vulnerabilities are highly likely to be detected with at most a few dozen test cases or less*

To provide a better view of the comparison between the two input generation techniques, we produced a set of plots. All of them are available in the appendix. Figures 3 and 4 depict the results when the subjects were protected with a WAF. The box-plots depict the results of μ**4SQLi** (recall it is non-deterministic) in terms of $\%T$ (lower part) and $\%T_e$ (upper part). The dash and triangle dots are the result of **Std**. As we can see, without having to resort to a statistical test, the differences are clearly significant. In the upper part of the figures, none of the tests generated by **Std** could result in executable SQL statements and therefore missed all the vulnerabilities. By contrast, μ**4SQLi** missed only one of the vulnerabilities in **SugarCRM**. In short, from the figures and above tables, we can see that the performance of μ**4SQLi** in terms of generating tests that lead to flagged and executable SQL statements is significantly better than **Std**.

***RQ2**: Our mutation-based technique (μ**4SQLi**) generates a higher percentage of tests that can reveal SQLi vulnerabilities. Further, in the presence of a WAF, μ**4SQLi** is the only technique that is capable of doing so.*

4.5 Discussion

Results without the WAF indicate that both approaches can detect vulnerabilities in the examined subjects. Both techniques were able to provide, for most vulnerable parameters, test cases leading to SQL statements that are flagged by GreenSQL and deemed executable. It is interesting to note, however, that a significantly higher percentage of test cases generated flagged and executable statements when using μ**4SQLi**. The practical implications of these results is that, since the execution time of a test case generated by either **Std** or μ**4SQLi** is comparable, when testing many services with many input parameters, μ**4SQLi** will be a more effective and less costly technique to detect exploitable vulnerabilities. They will be more likely to be detected within a fixed test budget and will be detected faster.

Results with the WAF are even more dramatic. Only μ**4SQLi** is able, for all parameters but one, to generate flagged and executable SQL statements. Since the presence of a WAF or similar protection mechanism is a much more realistic situation in practice, these results imply that in many situations, standard attacks

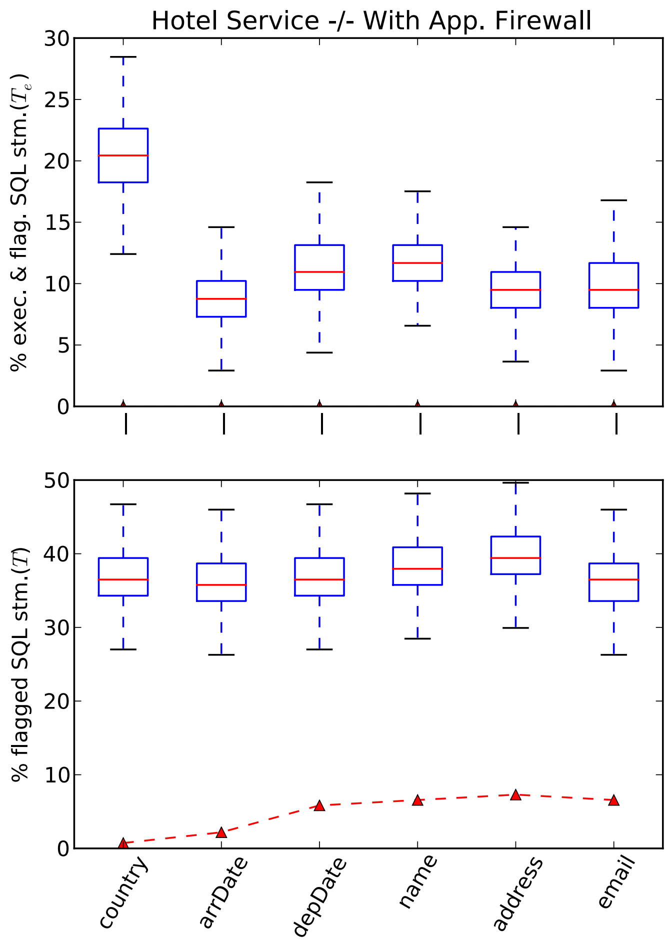

Figure 3: Results obtained from **HotelRS** with firewall enabled: the box-plots depict the results of μ**4SQLi**, the dashed line depicts the results of **Std**. None of the executable SQL statements generated by **Std** can get through the WAF.

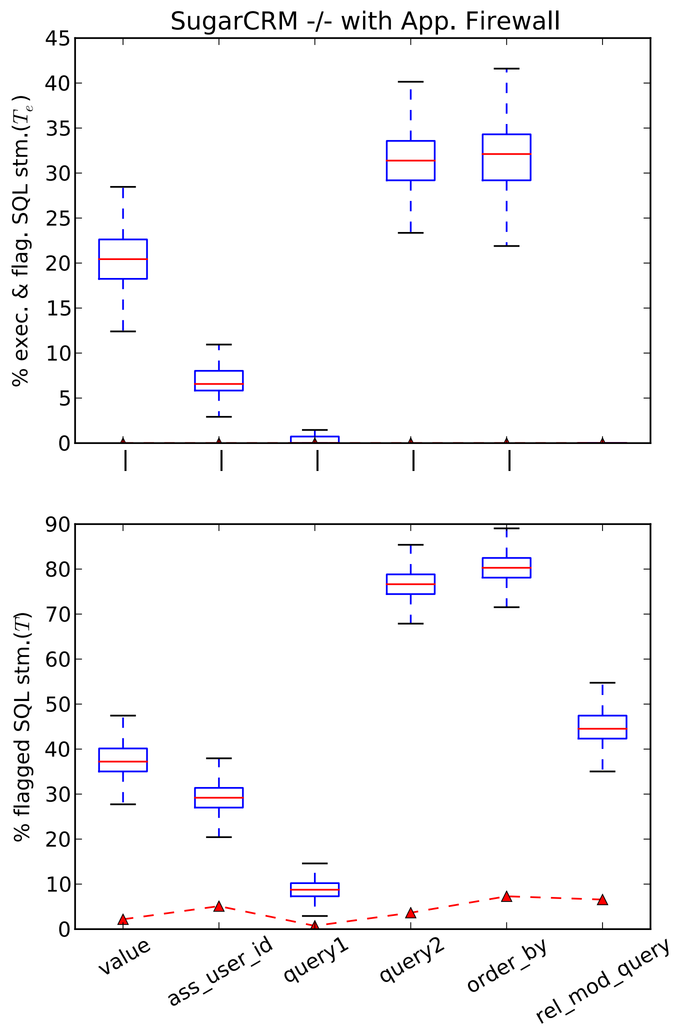

Figure 4: Results obtained from **SugarCRM** with the firewall enabled.

are not effective when looking for tangible evidence that there are *exploitable* SQLi vulnerabilities.

When the subjects were protected by the WAF, there was an even more contrasting difference in the results of the techniques. With the WAF enabled, μ**4SQLi** achieved results that are similar to when no WAF was used: T and T_e experienced only a slight drop. That difference was due to the WAF identifying and blocking only a small number of attacks. This is an evidence that the proposed obfuscation mutation operators are effective at bypassing the WAF. On the contrary, the test results for **Std** dropped considerably. T experienced a large drop and T_e went down to zero. This can be attributed to the WAF recognising most of the test cases as SQLi attacks and blocking them. The low percentage of test cases which bypass the firewall do not result in executable SQL statements.

Overall, the results indicate that the obfuscation and syntax-repairing have helped μ**4SQLi** in bypassing the WAF and triggering executable SQLi attacks.

4.6 Threats to Validity

The potential threats to validity of our results fall into the internal and external categories:

Internal threats: This is about whether the associations we observed between treatments (test techniques) and generated executable SQL statements can be confidently interpreted as due to the inherent properties of the techniques. For **Std** we used a comprehensive list of 137 known attack patterns mentioned in [1]. As far as we are aware, this is the state of the art for penetration testing in practice. Regarding μ**4SQLi**, since it is non-deterministic and to account for randomness, we generated and ran 1000 tests per parameter and then sampled (with replacement, a procedure called Bootstrapping) 10K test suites of 137 test cases to enable a statistical comparison with standard attacks. We have also inspected the reports of GreenSQL to remove any false alarms.

We chose ModSecurity as a WAF and used the OWASP Core Rule Set. This is a popular setting in practice used in many production systems.

External threats: This concerns the generalization of the results. Obviously, like any study in specific systems, it needs to be replicated. The computation cost of running such experiments is however high and, although we only used two systems in the experiments, them are from different domains and **SugarCRM** is used by real users as the number of downloads indicates. Although we compared only two test techniques, they are representative of the state of the art in black-box SQLi testing, as the review of related works indicates.

5 Conclusion

SQL injections have been ranked among the most common categories of vulnerabilities. Attacks that exploit such type of vulnerabilities increase rapidly over time. Automated testing techniques are important, not only to detect vulnerabilities in web services before they can be published, but also to reduce testing effort in contexts where the numbers of services and their input parameters are large. In particular, there is a need for black-box techniques that do not require access to the source code, as this is a common constraint when third party components are used or software development is (partly) outsourced. Existing techniques that have investigated this specific problem are bounded to known attack patterns that become out-dated very quickly, especially given the fast evolution of web services and their underpinning technologies. Their performance may also be limited by the presence of application protection mechanisms, such as firewalls, which may block known attacks. Our results confirm this problem by showing that state-of-practice, standard attacks do not, in most cases, make it through the firewall. In addition, the few that were not blocked by the firewall lead to non-executable SQL statements because of syntax errors.

We presented in this report an automated mutation technique for SQL injection vulnerabilities, supported by a tool, which focuses on mutating the input values of web service parameters. This technique makes use of a set of mutation operators that are able (1) to generate inputs with a high likelihood of modifying the behaviour of services, (2) to correct inputs to remove possible syntax errors due to mutations, and (3) to obfuscate attacks to increase their chances to make it though the firewall. The ultimate goal of our technique is to generate randomised inputs to detect SQL vulnerabilities by the way of SQL statements that are executable, are passing the firewall, and are unduly revealing or compromising data in the database. Our experimental results have demonstrated that our technique and tool performed much better than state-of-practice standard attack patterns, and that the probability of detecting SQL injection vulnerabilities is high, even in the presence of a firewall, and with a reasonable number of test case executions for each input parameter in each service.

Acknowledgement

This work was supported by the National Research Fund, Luxembourg (grant FNR/P10/03 and FNR4800382). We specially thank the testing and security team of our industry partner, CETREL, for their collaboration within this project. [11]

References

[1] N. Antunes, N. Laranjeiro, M. Vieira, and H. Madeira. Effective detection of SQL/XPath injection vulnerabilities in web services. In *Proceedings of the 6th IEEE International Conference on Services Computing (SCC '09)*, pages 260–267, 2009.

[11] http://www.cetrel.lu

[2] N. Antunes and M. Vieira. Detecting SQL injection vulnerabilities in web services. In *Proceedings of the 4th Latin-American Symposium on Dependable Computing (LADC '09)*, pages 17–24, 2009.

[3] D. Appelt, N. Alshahwan, and L. Briand. Assessing the impact of firewalls and database proxies on sql injection testing. In *Proceedings of the 1st International Workshop on Future Internet Testing*, 2013.

[4] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini. Ws-taxi: A wsdl-based testing tool for web services. In *ICST*, pages 326–335, 2009.

[5] T. Beery and N. Niv. Web application attack report, 2013.

[6] A. Ciampa, C. A. Visaggio, and M. Di Penta. A heuristic-based approach for detecting SQL-injection vulnerabilities in web applications. In *Proceedings of the ICSE Workshop on Software Engineering for Secure Systems (SESS '10)*, pages 43–49, 2010.

[7] J. Coffey, L. White, N. Wilde, and S. Simmons. Locating software features in a soa composite application. In *Web Services (ECOWS), 2010 IEEE 8th European Conference on*, pages 99–106, 2010.

[8] M. Cova, V. Felmetsger, and G. Vigna. Vulnerability analysis of web-based applications. In L. Baresi and E. Nitto, editors, *Test and Analysis of Web Services*, pages 363–394. Springer Berlin Heidelberg, 2007.

[9] B. Efron and R. Tibshirani. *An Introduction To The Bootstrap*, volume 57. CRC press, 1993.

[10] I. A. Elia, J. Fonseca, and M. Vieira. Comparing sql injection detection tools using attack injection: An experimental study. In *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering (ISSRE '10)*, pages 289–298, 2010.

[11] J. Fonseca, M. Vieira, and H. Madeira. Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks. In *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing (PRDC '07)*, pages 365–372, 2007.

[12] M. Fossi and E. Johnson. Symantec global internet security threat report, volume xiv, 2009.

[13] X. Fu and K. Qian. SAFELI: SQL injection scanner using symbolic execution. In *Proceedings of the workshop on Testing, Analysis, and Verification of Web Services and Applications (TAV-WEB '08)*, pages 34–39, 2008.

[14] W. G. Halfond, J. Viegas, and A. Orso. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE '06)*, pages 13–15, 2006.

[15] W. G. J. Halfond and A. Orso. Amnesia: analysis and monitoring for neutralizing SQL-injection attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*, pages 174–183, 2005.

[16] W. G. J. Halfond and A. Orso. Preventing SQL injection attacks using AMNESIA. In *Proceedings of the 28th International Conference on Software Engineering (ICSE' 06)*, pages 795–798, 2006.

[17] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Proceedings of the 21st Usenix Security Symposium*, 2012.

[18] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai. Web application security assessment by fault injection and behavior monitoring. In *Proceedings of the 12th International Conference on World Wide Web (WWW '03)*, pages 148–159, 2003.

[19] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[20] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, pages 199–209, 2009.

[21] I. Lee, S. Jeong, S. Yeo, and J. Moon. A novel method for SQL injection attack detection based on removing SQL query attribute values. *Mathematical and Computer Modelling*, 55(1):58–68, 2012.

[22] R. Sekar. An efficient black-box technique for defeating web application attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium*, 2009.

[23] H. Shahriar and M. Zulkernine. MUSIC: Mutation-based SQL injection vulnerability checking. In *Proceedings of the 8th International Conference on Quality Software (QSIC'08)*, pages 77–86. IEEE, 2008.

[24] L. K. Shar, H. B. K. Tan, and L. Briand. Mining sql injection and cross site scripting vulnerabilities using hybrid program analysis. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 642–651, 2013.

[25] Y. Shin. Improving the identification of actual input manipulation vulnerabilities. In *Proceedings of the 14th ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2006.

[26] Y. Shin, L. Williams, and T. Xie. Sqlunitgen: Test case generation for sql injection detection. *North Carolina State University, Raleigh Technical report, NCSU CSC TR*, 21, 2006.

[27] B. Smith, L. Williams, and A. Austin. Idea: using system level testing for revealing SQL injection-related error message information leaks. In *Proceedings of the 2nd International Conference on Engineering Secure Software and Systems (ESSoS '10)*, pages 192–200, 2010.

[28] SQL Injection Wiki. SQL injection cheat sheet. http://www.sqlinjectionwiki.com/, 2013.

[29] The Open Web Application Security Project (OWASP). Testing for SQL injection (owasp-dv-005). http://www.owasp.org, 2013.

[30] M. Vieira, N. Antunes, and H. Madeira. Using web security scanners to detect vulnerabilities in web services. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems Networks (DSN '09)*, pages 566–571, 2009.

[31] W3C. Character entity references in HTML 4. http://www.w3.org/TR/html4/sgml/entities.html, 2012.

[32] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, pages 32–41, 2007.

[33] K. Wei, M. Muthuprasanna, and S. Kothari. Preventing SQL injection attacks in stored procedures. In *Proceedings of the Australian Software Engineering Conference (ASWEC '06)*, pages 191–198, 2006.

[34] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

Appendix

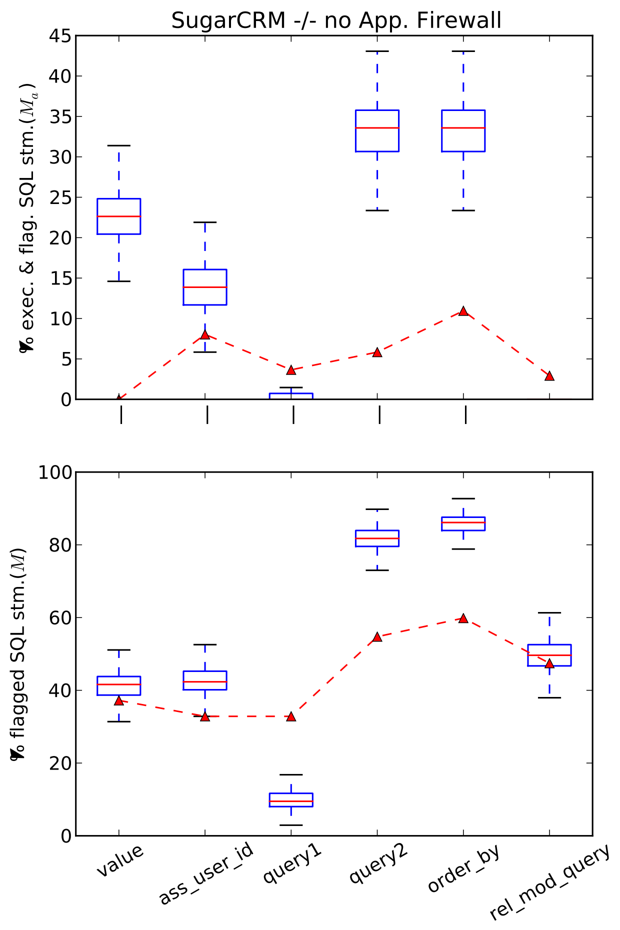

Figure 5: Results obtained from **SugarCRM** when no WAF was used. The box-plots depict the results of μ**4SQLi**, the dashed line depicts the results of **Std**. The box-plots are higher than the line, meaning that μ**4SQLi** performs better than **Std**.

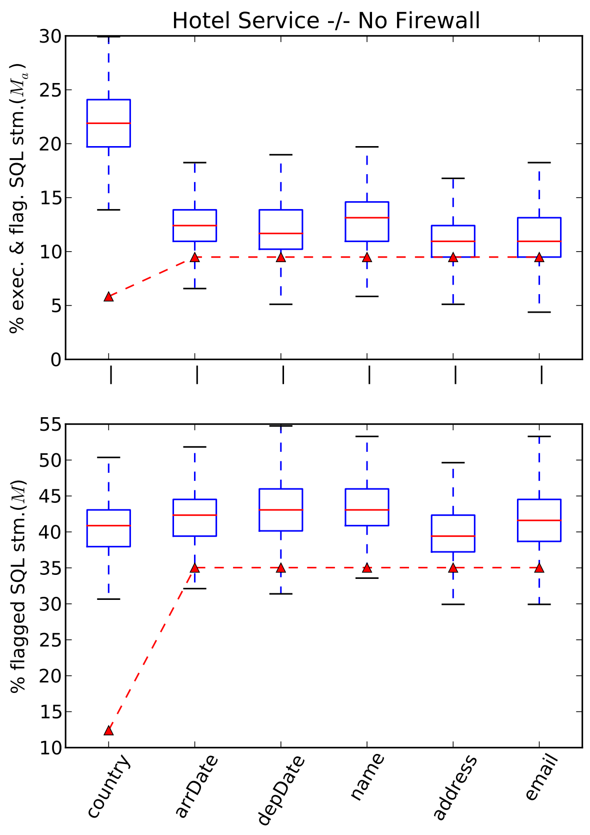

Figure 6: Results obtained from **HotelRS** when no WAF was used: the box-plots depict the results of μ**4SQLi**, the dashed line depicts the results of **Std**.