Behind an Application Firewall, Are We Safe from SQL Injection Attacks?

Dennis Appelt, Cu D. Nguyen, Lionel Briand
Interdisciplinary Centre for Security, Reliability and Trust (SnT Centre)
University of Luxembourg, Luxembourg
{dennis.appelt,duy.nguyen,lionel.briand}@uni.lu

Abstract—**Web application firewalls are an indispensable layer to protect online systems from attacks. However, the fast pace at which new kinds of attacks appear and their sophistication require that firewalls be updated and tested regularly as otherwise they will be circumvented. In this paper, we focus our research on web application firewalls and SQL injection attacks. We present a machine learning-based testing approach to detect holes in firewalls that let SQL injection attacks bypass. At the beginning, the approach can automatically generate diverse attack payloads, which can be seeded into inputs of web-based applications, and then submit them to a system that is protected by a firewall. Incrementally learning from the tests that are blocked or passed by the firewall, our approach can then select tests that exhibit characteristics associated with bypassing the firewall and mutate them to efficiently generate new bypassing attacks. In the race against cyber attacks, time is vital. Being able to learn and anticipate more attacks that can circumvent a firewall in a timely manner is very important in order to quickly fix or fine-tune the firewall. We developed a tool that implements the approach and evaluated it on *ModSecurity*, a widely used application firewall. The results we obtained suggest a good performance and efficiency in detecting holes in the firewall that could let SQLi attacks go undetected.**

I. Introduction

As the world wide web has been constantly evolving, many business sectors, such has e-banking, online shopping, e-government, and social networking, have made their services available on the web. Moreover, the adoption of cloud-based systems and services further accelerates this shift. However, this also caused the web to become a main target for malicious attackers. Recent studies found that the number of reported web vulnerabilities is growing sharply [10]. The latest statistics show that web applications experience up to 27 attacks per minute [7].

To protect valuable business and customer data, it is a good security practice to deploy layers of security protection, from low level intrusion detection mechanisms that work on network packages to application protections that are aware of domain specific business data and protocols. In the upper protection layer, Web Application Firewalls (WAFs) are an indispensable mechanism that is tailored to prevent specific types of attacks, such as SQL Injection or Cross-Site Scripting. When deployed, a WAF examines every request submitted to a target system to decide whether it is legitimate or malicious. The WAF makes this decision by examining each input entry in the request and checking whether or not the value matches an attack pattern, typically using a set of rules (e.g., regular expressions).

However, the constant evolution of attacks and their sophistication require that WAFs be updated and tested regularly as otherwise they will be circumvented. This is a laborious and costly task: a security expert needs, at the same time, to learn and identify prominent attacks that might happen to systems under protection, and to be able to fine-tune WAFs so that they block those attacks, yet preventing false positives from blocking legitimate requests.

In this paper we focus on web application firewalls that aim at preventing SQL injection (SQLi) attacks. Our goal is to develop automated testing techniques that generate SQLi attacks bypassing the WAFs and therefore help uncover holes in WAFs. We develop a SQLi grammar based on known SQLi attacks to date and an automated input generation technique that makes use of the grammar to automatically generate attack payloads (or tests, for brevity), which can be seeded into inputs of web-based applications that are protected by a firewall. Moreover, by using machine learning, our technique can learn incrementally from the tests that are blocked or passed by the firewall and then, based on characteristics that seem to be associated with bypassing the firewall, select tests matching those characteristics and mutate them to efficiently generate even more tests with high chance to bypass the firewall. Through such an iterative process, we hope to effectively and efficiently uncover most SQLi vulnerabilities. As a result, given a firewall under test, our approach can quickly and automatically provide the security expert with a set of various SQLi attacks that can circumvent the firewall to reach the systems under protection. This is an invaluable input that can help fix the firewall so that further attacks are prevented.

We have developed a tool that implements the approach and evaluated it with *ModSecurity*[1], a popular and representative web application firewall, which we configured to protect a web service application from SQLi attacks. We evaluated our proposed approach and found that it significantly outperforms a random test case generation approach. It generated significantly more distinct SQLi attacks bypassing the WAF, at a faster pace.

The remainder of the paper is structured as follows. Section II provides background information on WAFs and SQLi attacks. Section III discusses in detail our approach, followed by Section IV where we introduce our experiments and results.

[1]https://www.modsecurity.org

Finally, Section V concludes this paper and discusses future work.

II. Background and Related Work

A. Background

SQL Injection Vulnerabilities. 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 a SQL statement is formed, special functions are used to send the SQL statement to the database server to be executed. For example, a SQL statement can be 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 input parameters that lack proper validation or sanitisation. An attacker might construct input values in a way that changes the behaviour of the resulting SQL statement and performs arbitrary actions on the database (e.g. 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* received the attack payload *' or 1=1 --*, the resulting SQL statement is:

```
select * from hotelList where country='' or 1=1 --'
```

The clause *or 1=1* is a tautology, i.e., the condition will always be true, and is thus able to bypassing the original condition in the *where* clause, making the SQL query return all rows in the table.

Web Application Firewalls. Web applications with high security requirements are commonly protected by WAFs. In the overall system architecture, a WAF is placed in front of the web application that has to be protected. Every request that is sent to the web application is examined by the WAF before it reaches the web application. The WAF hands over the request to the web application only if the request complies with the firewall's rule set.

A common approach to define the firewall's rule set is using a black-list. A black-list contains string patterns, typically defined as regular expressions. Requests recognised by these patterns are likely to be malicious attacks (e.g., SQLi) and, therefore, are blocked. For example, the following regular expression describes the syntax for SQL comments, e.g., /**/ or #, which are frequently used in SQLi attacks:

```
/\*!?|\*/|[';]--|--[\s\r\n\v\f]|(?:--[^-]*?-) |
                    ([^\-&])#.*?[\s\r\n\v\f]|;?\\x00
```

There are several reasons why a WAF may provide insufficient protection, including implementation bugs or misconfiguration. One way to ensure the resilience of a WAF against attacks is to rely on an automated testing procedure that thoroughly and efficiently detects vulnerabilities. This paper addresses this challenge for SQL injections, one of the main types of vulnerabilities in practice.

B. Related Work

Previous research on ensuring the resilience of IT systems against malicious requests has focused on the testing of firewalls as well as input validation mechanisms.

Offutt et al. introduced the concept of Bypass Testing in which an application's input validation is tested for robustness and security [18]. Tests are generated to intentionally violate client-side input checks and are then sent to the server application to test whether the input constraints are adequately evaluated. Liu et al. proposed an automated approach to recover an input validation model from program source code and formulated two coverage criteria for testing input validation based on the model [16]. In contrast, we propose a black-box technique that does not require access to source code or client-side input checks to generate test cases. In our approach we use machine learning to identify the patterns recognised by the firewall as SQLi attacks and generate bypassing test cases that avoid those patterns.

The topic of testing network firewalls is addressed by an abundant literature. Although network firewalls operate on a lower layer than application firewalls, which are our focus, they share some commonalities. Both use policies to decide which traffic is allowed to pass or should be rejected. Therefore, testing approaches to find flaws in network firewall policies might also be applicable to web application firewall policies. Bruckner et al. proposed a model-based testing approach which transforms a firewall policy into a normal form [8]. Based on case studies they found that this policy transformation increases the efficiency of test case generation by at least two orders of magnitude. Hwang et al. defined structural coverage criteria of policies under test and developed a test generation technique based on constraint solving that tries to maximise structural coverage [14]. Other research has focused on testing the firewalls implementation instead of policies. Al-Shaer et al. developed a framework to automatically test if a policy is correctly enforced by a firewall [1]. Therefore, the framework generates a set of policies as well as test traffic and checks whether the firewall handles the generated traffic correctly according to the generated policy. Some authors have proposed specification-based firewall testing. Jürjens et al. proposed to formally model the tested firewall and to automatically derive test cases from the formal specification [15]. Senn et al. proposed a formal language for specifying security policies and automatically generate test cases from formal policies to test the firewall [19]. In contrast, in addition to targeting application firewalls, our approach does not rely in any models of security policies or the firewall under test, such formal models are rarely available in practice.

III. Approach

We consider SQLi attack strings as small "programs" that aim at changing the intent of target SQL statements when they are concatenated to such programs. Therefore, we systematically surveyed existing SQLi attacks published in the literature, e.g., [11], [4], [3] and from the Internet (e.g., OWAPS[2], and SqlMap[3]. We then defined a context-free grammar for SQLi attacks and developed two attack generation techniques, random (RAN) and machine learning driven (ML-Driven), the latter being inspired by genetic programming [6], with the objective of generating tests (attacks) that can bypass web application firewalls.

A. A Context-Free Grammar for SQLi Attacks

We consider three main categories of SQLi attacks: *BOOLEAN*, *UNION*, and *PIGGY* that aim at manipulating the intended logic, exploiting the family of union queries, or injecting additional statements in the original SQL queries [11]. We define a grammar that covers three different contexts in which SQLi attacks can be inserted into: *numericContext*, *sQuoteContext*, and *dQuoteContext*. The first one targets SQL statements whose conditional parts (WHERE clauses) involve numeric data fields (e.g., *age*, *price*); the second and third target SQL statements which involve string data fields. The grammar is defined in the Extended Backus Normal Form, An excerpt of the grammar is listed as follows, in which *<ATTACK>* is the start symbol, "::=" is the production symbol, "," is concatenation, and "|" represents alternatives.

⟨*ATTACK*⟩ ::= ⟨*numericContext*⟩ | ⟨*sQuoteContext*⟩
  | ⟨*dQuoteContext*⟩ ;

⟨*numericContext*⟩ ::= ⟨*digitZero*⟩, ⟨*wsp*⟩, ⟨*booleanAttack*⟩, ⟨*wsp*⟩
  | ⟨*digitZero*⟩, ⟨*parC*⟩, ⟨*wsp*⟩, ⟨*booleanAttack*⟩, ⟨*wsp*⟩, ⟨*opOr*⟩, ⟨*parO*⟩, ⟨*digitZero*⟩
  | ⟨*digitZero*⟩, [⟨*parC*⟩], ⟨*wsp*⟩, ⟨*sqliAttack*⟩, ⟨*cmt*⟩ ;

⟨*sQuoteContext*⟩ ::= ⟨*squote*⟩, ⟨*wsp*⟩, ⟨*booleanAttack*⟩, ⟨*wsp*⟩, ⟨*opOr*⟩, ⟨*squote*⟩
  | ⟨*squote*⟩, ⟨*parC*⟩, ⟨*wsp*⟩, ⟨*booleanAttack*⟩, ⟨*wsp*⟩, ⟨*opOr*⟩, ⟨*parO*⟩, ⟨*squote*⟩
  | ⟨*squote*⟩, [⟨*parC*⟩], ⟨*wsp*⟩, ⟨*sqliAttack*⟩, ⟨*cmt*⟩ ;

⟨*dQuoteContext*⟩ ::= ⟨*dquote*⟩, ⟨*wsp*⟩, ⟨*booleanAttack*⟩, ⟨*wsp*⟩, ⟨*opOr*⟩, ⟨*dquote*⟩
  | ⟨*dquote*⟩, ⟨*parC*⟩, ⟨*wsp*⟩, ⟨*booleanAttack*⟩, ⟨*wsp*⟩, ⟨*opOr*⟩, ⟨*parO*⟩, ⟨*dquote*⟩
  | ⟨*dquote*⟩, [⟨*parC*⟩], ⟨*wsp*⟩, ⟨*sqliAttack*⟩, ⟨*cmt*⟩ ;

⟨*sqliAttack*⟩ ::= ⟨*unionAttack*⟩ | ⟨*piggyAttack*⟩ | ⟨*booleanAttack*⟩ ;

⟨*unionAttack*⟩ ::= ... ; ⟨*piggyAttack*⟩ ::= ... ; ⟨*booleanAttack*⟩ ::= ⟨*orAttack*⟩ | ⟨*andAttack*⟩ ;

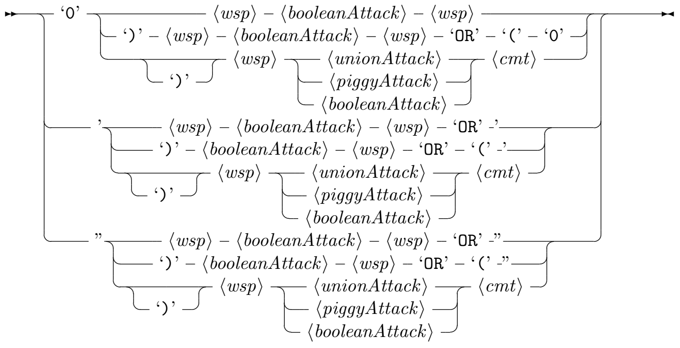
⟨*opNot*⟩ ::= `!` | `not` ; ⟨*opBinInvert*⟩ ::= `~` ; ⟨*opEqual*⟩ ::= `=` ; ⟨*opLt*⟩ ::= `<` ; ⟨*opGt*⟩ ::= `>` ; ⟨*opLike*⟩ ::= `like` ; ⟨*opIs*⟩ ::= `is` ; ⟨*opMinus*⟩ ::= `-` ; ⟨*opOr*⟩ ::= `or` | `||` ; ⟨*opAnd*⟩ ::= `and` | `&&` ; ⟨*opSel*⟩ ::= `select` ⟨*opUni*⟩ ::= `union` ; ⟨*opSem*⟩ ::= `;` ;

⟨*cmt*⟩ ::= `#` | ⟨*ddash*⟩, ⟨*blank*⟩; ⟨*ddash*⟩ ::= `--`

⟨*inlineCmt*⟩ ::= `/**/` ; ⟨*blank*⟩ ::= ␣ ; ⟨*wsp*⟩ ::= ⟨*blank*⟩ | ⟨*inlineCmt*⟩

This grammar can be extended to incorporate further forms of attacks. For example, the non-terminal *<blank>* can have more semantically equivalent terminal characters: +, /**/, or unicode encodings: %20, %09, %0a, %0b, %0c, %0d and %a0; the quotes (single or double) can be represented using HTML encoding, and so on.

We represent the grammar in the syntax diagram depicted in Fig. 1. Due to the limited space reason we only unpack the grammar up to the attack levels (BOOLEAN, UNION, and PIGGY). The diagram reads from the entry point on the left and ends at the end point on the right. The paths between these two points are alternative productions that can be produced or recognised by the grammar.



'0'
⟨wsp⟩ – ⟨booleanAttack⟩ – ⟨wsp⟩
')' – ⟨wsp⟩ – ⟨booleanAttack⟩ – ⟨wsp⟩ – 'OR' – '(' – '0'
')'
⟨wsp⟩
⟨unionAttack⟩
⟨piggyAttack⟩
⟨booleanAttack⟩
⟨cmt⟩
'
⟨wsp⟩ – ⟨booleanAttack⟩ – ⟨wsp⟩ – 'OR' -'
')' – ⟨booleanAttack⟩ – ⟨wsp⟩ – 'OR' – '(' -'
"
⟨wsp⟩ – ⟨booleanAttack⟩ – ⟨wsp⟩ – 'OR' -"
')' – ⟨booleanAttack⟩ – ⟨wsp⟩ – 'OR' – '(' -"

Fig. 1. The syntax diagram of the proposed grammar: the entry point is on the left, the exit point is on the right.

B. Random (RAN) Generation

Based on the syntax diagram shown in Fig. 1, the random generation (RAN) procedure is straightforward: we start from the entry point, and at any branching point we select randomly an alternative to follow. Since there is no loop, we will always reach the end point. The output SQLi attack is produced by concatenating the terminal symbols we have encountered along the path.

In order to produce a set of diverse random SQLi attacks that yield a good coverage of the different rules of the grammar, an alternative branch is selected when traversing the syntax diagram (graph) with a probability proportional to the number of distinct paths through which the branch can reach the end point. So for instance, if there are two alternative branches at a branching point, and if following the first one will give us three paths to reach the end point, while following the second will give us four paths to the end point, the selection probabilities of the first and second alternatives are 3/7 and 4/7, respectively.

RAN is equivalent to our previous work on mutation-based input generation [5] with respect to getting attacks to bypass WAFs. The mutation operators presented in [5] are similar to the production rules of the grammar, including rules specified for syntax-repairing and obfuscation.

[2]https://www.owasp.org
[3]http://sqlmap.org

C. ML-Driven Generation

Our approach, called ML-Driven, is inspired by genetic programming and search-based test generation [6], [17], [2]. We face the problem to efficiently choose from a large set of SQLi attacks the ones that are more likely to reveal holes in the WAF under consideration. The problem is challenging because there is little information available to calculate how close a test comes to bypassing the WAF. When a test is executed only one of the following two events can be observed: *bypassing*, or *blocked*. This leaves the search with no guidance to effectively assess how close a blocked attack is from bypassing the WAF. To tackle the problem, we use machine learning to model how the elements (features of attacks) of the tests are associated with high likelihoods of bypassing the WAF. In the search process, tests that are predicted to have such high likelihood are considered to have a high fitness and are likely candidates for mutation.

More specifically, our approach employs, first, the random test generation technique to generate some initial tests. These tests are sent to web applications protected by the WAF. Depending on whether they bypass or are blocked by the WAF, they are labelled as "P" or "B", respectively. We encode these tests and use them as initial training data to learn a model predicting the likelihood (f) with which tests can bypass the WAF. Using this measure we can rank, select, and mutate tests associated with high f values to produce new tests. These new tests are then executed, and their results ("P" or "B") are used to improve the prediction model, which will in turn help generating more distinct tests that bypass the WAF.

In what follows, we will discuss in detail how tests are decomposed and encoded for machine learning and the mutation process, which we use to generate new SQLi attacks. Finally, we describe our overall ML-driven test generation approach that aims at iteratively finding new and effective attacks.

1) Test Decomposition: We can derive tests from the grammar by applying recursively its production rules. Such a procedure is represented as derivation trees. A derivation tree (also called parse tree) of a test is a graphical representation of the derivation steps that are involved in producing the test. In a derivation tree, an intermediate node presents a non-terminal symbol, a leave node represents a terminal one, and edges are derivations. Fig. 2 depicts the derivation tree of the BOOLEAN attack test: *'␣OR"a"="a"#*. In the course of generating this test, we first apply the *<ATTACK>* rule:

$$\langle ATTACK \rangle ::= \langle numericContext \rangle \mid \langle sQuoteContext \rangle \mid \langle dQuoteContext \rangle \ ;$$

and derive *<sQuoteContext>*. We then apply the third rule of the grammar to derive *<squote>*, *<wsp>*, *<sqliAttack>*, and *<cmt>*. This procedure is repeated until all the leave nodes are terminal symbols.

We make use of derivation trees to identify which parts of a SQLi attack are likely to be responsible for the attack being blocked or passing. Specifically, for each test, we decompose its derivation tree into *slices*, which are defined as follows:

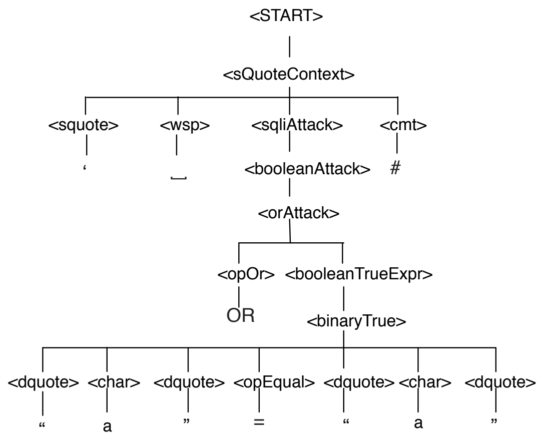


Fig. 2. The derivation tree of the "boolean" SQLi attack: *'␣OR"a"="a"#*.

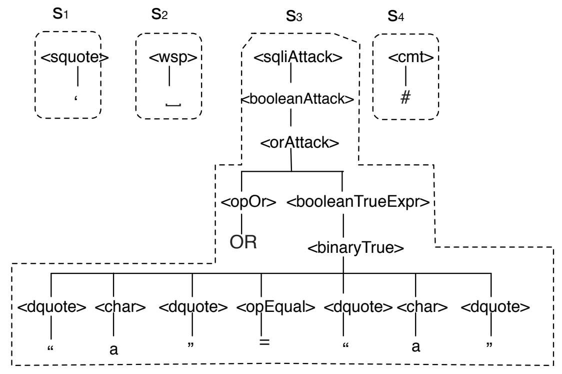


Fig. 3. Example of some slices decomposed from the tree in Fig. 2.

Definition 1 (Slice)**.** A slice s of a derivation tree T is a subtree of T such that the root of s is a non-terminal node of T, except those that represent *<ATTACK>*, *<numericContext>*, *<sQuoteContext>*, and *<dQuoteContext>*.

We skip the start symbol and its children because subtrees extracted from them will be (closely) equivalent to the original derivation tree. Such decompositions provide no or little information as to why a test is blocked or bypassing.

Definition 2 (Minimal Slice)**.** A slice s is minimal if it has only two nodes: a root, and its single child is a terminal symbol.

The procedure for decomposing a tree is detailed in Algorithm 1. We skip the start symbol and its child (*context*), and then get slices from the descendants of the child, recursively.

Applying the decomposition procedure to the derivation tree in Fig. 2, we obtained a set of 12 distinct slices. Fig. 3 shows a sample of four slices decomposed from the tree.

We conjecture that the appearance of one or more slices in a test could result in the test getting blocked or bypassing. In the next sections we develop this idea further in analysing slices of a collection of tests and predict, using machine learning, how their appearance in the tests affect their likelihood of bypassing a WAF or being blocked.

2) Encoding Tests for Machine Learning: Given a set of tests that have been labelled with their execution result against

Algorithm 1 Test decomposition into slices.

```
1:  procedure DECOMPOSE($inputTree$)
2:      $S \leftarrow \emptyset$
3:      $context \leftarrow inputTree.child$                ▷ the only child
4:      for all $child \in context$ do
5:          VISIT($child$, $S$)
6:      end for
7:      return $S$
8:  end procedure
9:  procedure VISIT($node$, $S$)
10:     $s \leftarrow sliceFrom(node)$                ▷ get a sub-tree from node
11:     if $s \notin S$ then
12:         $S \leftarrow s$
13:     end if
14:     if s is minimal then
15:         return
16:     else
17:         for all $child \in s$ do
18:             VISIT($child$, $S$)
19:         end for
20:     end if
21: end procedure
```

a WAF, that is a “P” or “B” label, we transform each test into an observation instance to feed our machine learning algorithm.

1) Each test is decomposed into a vector of slices $t_i = \langle s_1, s_2, \ldots, s_{N_i} \rangle$ by applying the test decomposition procedure.
2) Each slice is assigned a globally unique identifier. If the same slice is part of multiple tests, it is referenced by the same identifier. We map each unique slice to an attribute (a feature) of the output data corpus for machine learning.
3) Every test is transformed into an observation in the data corpus by checking whether the slices used as attributes are present or not in the corresponding vector of slices of the test.

As a concrete example, we have three tests t_1, t_2, t_3; the first two are blocked while the last can bypass a WAF. Their decompositions into slices and labels are shown on the left side of Table I, and their encoded presentation on the right side of the table. In total, we have five different slices from all the tests and they become attributes of the training data set for machine learning. If a slice appears in a test, its corresponding attribute value in the training data is “1”, and otherwise “0”.

TABLE I
AN EXAMPLE OF TEST DECOMPOSITIONS AND THEIR ENCODING.

t.id	vector	label
1	$\langle s_1, s_2, s_3 \rangle$	B
2	$\langle s_1, s_2, s_4 \rangle$	B
3	$\langle s_4, s_5 \rangle$	P

t.id	s_1	s_2	s_3	s_4	s_5	clz
1	1	1	1	0	0	B
2	1	1	0	1	0	B
3	0	0	0	1	1	P

3) ML-Driven Test Generation: By decomposing tests into slices and transforming them into observation data, we can now apply a machine learning technique to predict which slices or which compositions of slices are associated with tests bypassing a WAF. Since we expect the number of tests (aka attacks) that are blocked to be much greater than those that can bypass a WAF in practice (as otherwise the WAF is ineffective), our training data is imbalanced. This means that the number of observations with a “B” label is much greater than for the “P” label. As a result, we need a machine learning algorithm that can deal with imbalanced data. Specifically, we use the *RandomTree* classifier (initially proposed in [13]) that is implemented in the Weka suite [12]. When performing a preliminary study about alternative classification methods implemented in the Weka suite, we found that *RandomTree* performs better than other candidates in terms of dealing with imbalanced data, thus leading to better precision and recall. More importantly, the output of the algorithm is a decision tree whose structure can be easily exploited in our approach, as described below.

Algorithm 2 ML-Driven SQLi attack generation.

```
1:  procedure MLDRIVENGEN($initTests$, $outputTests$)
2:      $execute(initTests)$
3:      $trainData \leftarrow transform(initTests)$
4:      $DT \leftarrow learnClassifier(trainData)$      ▷ learn the initial
    classifier
5:      $currTests \leftarrow initTests$
6:      while not-done do
7:          $rankTests(currTests, DT)$
8:          repeat
9:              $t \leftarrow selectATest(currTests)$
10:             $V \leftarrow getSliceVector(t)$
11:             $pathCondition \leftarrow getPath(V, DT)$
12:             $s \leftarrow pickASliceFrom(V)$
13:             while $s \neq null$ do
14:                 if $satisfy(s, pathCondition)$ then
15:                     $newTests \leftarrow mutate(t, s, MAX_M)$
16:                     $currTests \leftarrow currTests \cup newTests$
17:                 end if
18:                 $s \leftarrow pickASliceFrom(V)$
19:             end while
20:         until $shouldUpdClassifier(currTests)$
21:         $execute(currTests)$                  ▷ new tests only
22:         $trainData \leftarrow transform(currTests)$
23:         $DT \leftarrow learnClassifier(trainData)$
24:     end while
25:     $outputTests \leftarrow filterByPassTests(currentTests)$
26:     return $outputTests$
27: end procedure
```

Algorithm 2 details the ML-Driven strategy. From an initial test set $initTests$, which can be generated by using the random strategy discussed in Section III-B or obtained from previous runs of this strategy, we first execute the tests against a target WAF, $execute(initTests)$, line 2. In fact, we have to execute only the tests for which the result against the WAF is not yet known. Then, the tests are transformed into the training set, $trainData \leftarrow transform(initTests)$, line 3. Details of this step are discussed in the previous section, III-C2. A classifier DT is then learned from the data using the *RandomTree* algorithm.

At line 6, the algorithm starts iterating through its main loop while the condition *not-done* holds. The steps within the loop are:

1) Ranking the current test set using the learned classifier, $rankTests(currentTests, DT)$, line 7. Tests are

ranked based on their prediction probability f to bypass the WAF.

2) Until the classifier should be updated, new mutants are generated as follows:
   - select the test with the highest ranking as candidate test for mutation, $t \leftarrow selectTests(currentTests)$, line 9. If more than one candidate has been ranked equally, the selection is random among them.
   - get the path condition (discussed below) from the classifier with the slice vector V of the test.
   - continuously pick a slice s from V, check whether it satisfies the determined path condition, $satisfy(s, pathCondition)$, and if it is the case, replace the slice with its alternatives, which also need to satisfy the path condition, to generate new tests $newTests \leftarrow mutate(t, s, MAX_M)$
3) Checking whether we should update the classifier, $shouldUpdClassifier(currTests)$, and then updating it if required.

Definition 3 (Path Condition)**.** A path condition is a predicate that characterises a path of a decision tree classifier. It is represented as a conjunction $\bigwedge_i^k (s_i = val)$, in which $val = 1 \mid 0$.

Given a decision tree and a slice vector V of a test t, we can obtain the path condition that t satisfies by visiting the tree from the root and check the presence (value = 1) or absence (value = 0) of the attributes encountered with respect to V. For instance, Fig. 4 presents a decision tree learned from the example data discussed in Table I. For test t_1, s_1 being present in its slice vector $\langle s_1, s_2, s_3 \rangle$, we go to the left branch since the path condition is $(s_1 = 1)$. Similarly, if we consider test t_3 with its slice vector $\langle s_4, s_5 \rangle$, when we visit the tree, we go to the right branch because s_1 is absent and its path condition is $(s_1 = 0)$.

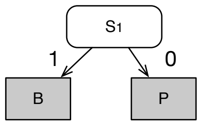


Fig. 4. An example of a decision tree obtained from the training data in Table I.

A slice is said to satisfy a path condition if it does not affect the truth value of the condition. That is, it either appears in the predicate and complies with it, or does not appear in the predicate. In our approach, we rank tests and select those that fall in a tree leave with a high likelihood of bypassing the WAF. Therefore, we consider the path conditions of those tests to be good indicators about test content that the WAF might ignore. As a result, in our mutation step, $newTests \leftarrow mutate(t, s, MAX_M)$, we select slices that do not appear in the condition of t and replace it with its equivalent alternative slices that also need to satisfy the condition. Take, for example, t_2 with its slice vector $\langle s_1, s_2, s_4 \rangle$. Since its path condition is $(s_1 = 1)$, we can select s_2 or s_4 and replace them with their alternatives.

Equivalent alternatives of a slice are determined based on the root non-terminal symbol of the slice and all production rules of the grammar that start with this symbol. For example, taking slice s_2 in Fig. 3 that starts with <wsp> and derives <blank>, we obtain only one production rule from the grammar:

$$\langle \textit{wsp} \rangle \quad ::= \langle \textit{blank} \rangle \mid \langle \textit{inlineCmt} \rangle \ ;$$

As a result, we determine only one alternative slice that starts with <wsp> and derives <inlineCmt>.

Along the test generation process, the classifier is updated regularly when the condition $shouldUpdClassifier$ holds. Currently we decide to update the classifier whenever we reach more than 4000 new tests. Then we select up to 6000 blocked tests, which are labelled as B, and all bypassing tests for retraining the classifier. This configuration was chosen so that our approach can still finish in a reasonable amount of time on a computer of 8Gb RAM and 2.8GHz duo core CPU.

There are several conditions that can drive the termination of the test generation process: (i) the time budget, (ii) the improvement of the decision classifier after a number of iterations, and (iii) the number of observed tests that can bypass the WAF under test. The second one computes the precision and recall values of the classifier and, if these do not change after a number of updates, we may conclude that no classifier improvement can be achieved, stop the update process, and use the classifier to generate as many new tests as time budget allows.

In the ML-Driven Test Generation strategy, MAX_M is an important variable that controls our approach to explore the test space either *broadly* or *deeply*. When MAX_M is small, the approach accounts for fewer alternative slices and generates more test cases, thus exploring the test space in a broader fashion. When MAX_M is large, the approach selects fewer tests and explores many more alternative slices, thus exploring the test space in a deeper fashion.

IV. Experiments and Results

In this work we have performed two sets of experiments to investigate (i) the evolution of the machine learning classifier, i.e., the improvement of the classifier during iterative training, (ii) the configuration of the machine learning algorithm that produces the best results, and (iii) the overall performance of the proposed ML-Driven test generation approach in terms of effectiveness and efficiency.

A. Subject Applications

In our experiments we select *ModSecurity* with the latest version[4] of the OWASP Core Rule Set[5] as a target WAF. *ModSecurity* is an open source web application firewall that can be deployed with Apache HTTP Server[6] to protect web applications hosted under the server. Depending on the applications under protection, different firewall rule sets defined for different purposes can be used. The rule set that we chose consists of defence rules against various kinds of attacks, including Trojan, Denial of Service, and SQL Injection.

[4]Version 2.2.9,as of Oct 14^{th} 2014
[5]https://github.com/SpiderLabs/owasp-modsecurity-crs
[6]http://httpd.apache.org

The web applications under protection are HotelRS, Cyclos, and SugarCRM. HotelRS is a SOA (service-oriented architecture) based system providing web services for room reservation and was developed and used in [9]. Cyclos is a popular open source Java/Servlet Web Application for e-commerce and online payment[7]. SugarCRM is a popular customer relationship management system[8]. SugarCRM and Cyclos have been widely used in practice. In our experiment setting, the three applications are deployed on an Apache HTTP Server under Linux. *ModSecurity* is embedded within the web server; it has to protect the applications' web services from SQLi attacks. Specifically, since these web services receive SOAP messages[9] from web clients, a malicious client can seed a SQLi attack string into a SOAP message and submit it to the web services in order to gain illegal access to data or functionality of the system.

In this paper, note that our testing target is the WAF that protects the applications, not the applications themselves, as our focus is on testing firewalls. HotelRS, SugarCRM, and Cyclos play solely the role of a destination for SQLi tests that bypass the WAF.

B. Procedure

We have implemented the proposed technique, ML-Driven, along our baseline RAN, and integrated them into our SQLi test tool, called *Xavier*. *Xavier* supports our objective of automated testing of web services for SQLi vulnerabilities and has been described in [5]. RAN or ML-Driven can generate tests in the form of SQLi attack strings, for example *'␣OR"1"="1"#*. *Xavier* takes such tests and injects them into SOAP message samples, turn them into malicious inputs, and send them to an application under test. *Xavier* relies on sample SOAP message as inputs. They can be taken from existing web service test suites, or can easily be generated from the WSDL[10] that describes the service under test.

In our experiments, from the WSDLs of the subject applications, we created nine SOAP messages corresponding to nine different web service operations. Running our experiments on such messages would have taken 45 days of computations on a single node and actually took five days on a limited access high performance cluster [20]. Given such computational constraints, these SOAP messages were selected very carefully in order to achieve diversity with respect to the number of XML elements and attributes, and the maximum level of nested XML elements. Each message consists of a number of parameters and their legitimate values, which the applications expect and that the WAF has to let through. In our testing process each SOAP message is considered separately. A test generation technique, RAN or ML-Driven, continuously generates attacks, injecting one attack each time into a randomly-chosen parameter of the SOAP message to create a new SOAP message, and sends the latter to the web server.

Incoming SOAP requests to the web server are first treated by the WAF and only those that comply with firewall rules are forwarded to web applications, otherwise they are blocked. In case a request is blocked, the WAF replies to the client who has issued the request with a special response, stating that the request has been denied. When our testing tool, *Xavier*, receives such a response, it marks the test, embedded in the original request, with a blocked label "B", and otherwise "P".

C. Variables

The following variables are controlled or measured in our experiments:
D_t: Number of distinct tests that can bypass a target WAF. We define distinct tests as tests that may be treated differently by the WAF. In other words, given two distinct tests, one might be caught by the WAF while the other bypasses it, even though they may belong to a same category. This may be caused by tests in a category that should be handled by different rules, some of them missing or incorrect in the current WAF rules set, or by an identical rule that is not general enough to block all tests in a category. Therefore, executing distinct tests is useful as, when testing a WAF, there is no a priori knowledge (e.g., proprietary WAF) about how they are handled by firewalls rules. In fact, determining whether bypassing tests belong to similar successful attack patterns, which may be due to the same WAF holes, is the objective of our machine learning strategy used to further guide the test process. Given the above observations, the number of distinct tests (D_t) is a valid measure of how effective a test generation technique is, at a given point in time, in generating different attacks that can bypass the WAF.
C: Wall-clock time to obtain a set of tests. This is a measure of cost, as given the size of the input space, we need to be as quick as possible in generating bypassing tests.
E: Efficiency of a technique. This is measured as the number of new bypassing tests generated per minute.
K: A parameter of the *RandomTree* algorithm that sets the percentage of attributes the algorithm must consider at each step during the construction of the classification tree.
$F{-}measure$: Standard prediction accuracy measure of a classifier, equal to the harmonic mean of its precision and recall.
M_{size}: Size of a classifier, measured as the number of nodes in the tree. This is an indication of the cost of classification.

D. Experiments and Results

1) Improvement of Classifiers: By default, the parameter K of the algorithm *RandomTree* is assigned automatically to $log_2(N) + 1$, where N is the total number of attributes of a training data set. However, in our pilot experiments, we realised that such a value for K yields classifiers with poor performance (very low $F{-}measure$ for class "P") with our training data. More generally, we observed that when K is increasing, until some thresholds are reached, $F{-}measure$ for

[7]http://project.cyclos.org
[8]http://sourceforge.net
[9]http://www.w3.org/TR/soap12-part1
[10]http://www.w3.org/TR/wsdl

class “P” is increasing and then is reaching a plateau beyond that. Therefore, our first question investigates the effect of K on the classifiers and what are recommended ranges for obtaining optimal results.

RQ1: *What is the best configuration of K for the* RandomTree *algorithm to generate a good classifier in terms of $F\!-\!measure$ and M_{size}?*

Beside $F\!-\!measure$, M_{size} is also an important quality of a classifier since it affects the cost of using the classifier. When M_{size} is larger, more computation time is required in ranking tests and checking path conditions. Therefore, ideally, we prefer smaller classifiers with high $F\!-\!measure$, that are not expensive to learn.

Since an accurate classifier is essential, as it drives our test generation approach to delivering better performance, our second research question studies how our classifier fares across training iterations, to determine how long it takes before they reach their optimal accuracy.

RQ2: *How do classifiers improve over training iterations in terms of $F\!-\!measure$?*

We ran ML-Driven with two values of MAX_M (10 and 100, more details are in the next section) and different values of K: 10%, 20%, . . ., 60% of the total number of the attributes (slices) of the training data at each iteration. Figure 5 depicts at the same time the average $F\!-\!measure$ for class “P” and the average M_{size} over iterations. We plot only data obtained with K equal to 20%, 40%, and 60% to avoid cluttering the figures. Plots corresponding to the other values of K share similar trends and, thus, do not affect our interpretation. Across iterations we find that higher K values tend to yield a higher $F\!-\!measure$ and a lower M_{size}. $F\!-\!measure$ and M_{size} corresponding to large K values seem to converge as their lines are very close to one another. In addition, though not shown on the figures, $F\!-\!measure$ for class “B”, which is the majority class in training data, remains constantly high (above 99%) regardless of K.

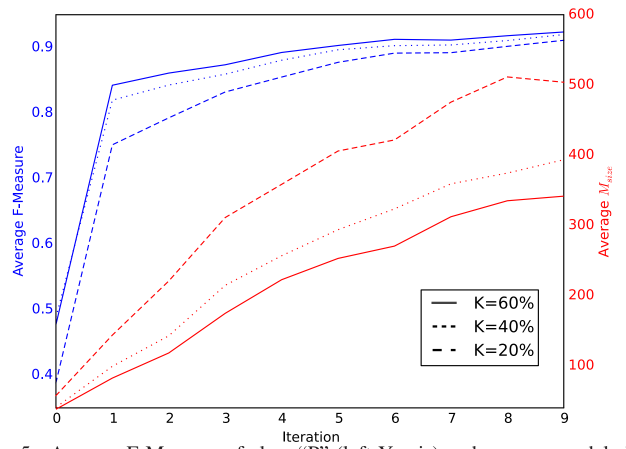


Fig. 5. Average F-Measure of class “P” (left Y-axis) and average model size (M_{size}, right Y-axis) for different K values over iterations. The data were obtained from 20 repetitions.

Based on these results we find that a K around 40% produces classifiers that are optimal in terms of $F\!-\!measure$ and M_{size} (**RQ1**). We should not increase K above that range since it would require more time for training the classifier while bringing very little accuracy improvement. Moreover, the results suggest that the accuracy of the classifier does significantly improve over iterations, especially in the first four or five iterations, though their size also grows steadily until iteration 10 or so (**RQ2**). To conclude, K values within the range 40%±10 should be used for *RandomTree* in our context and the training process should go, as a ballpark figure, through a minimum of 5 to 10 iterations. In all subsequent experiments we will only report results for K equal to 40%.

2) Performance Comparison: As discussed in Section III-C, the variable MAX_M controls ML-Driven to be broad (when MAX_M is small, by selecting more tests and mutating them with fewer alternative slices) or deep (MAX_M is large). Thus, in this experiment, we compare RAN and two variants of ML-Driven: ML-Driven B (broad, with $MAX_M = 10$) and ML-Driven D (deep, with $MAX_M = 100$).

RQ3: *Among ML-Driven B, ML-Driven D, and RAN, which one yields better performance in terms of the number of bypassing tests (D_t) and efficiency?*

To account for the degree of randomness involved in ML-Driven and RAN, we have repeated 10 times our experiment on each of the nine SOAP messages. In each run, ML-Driven B and ML-Driven D initially start by generating 2000 tests randomly. Each time a new test was generated, we have noted the passing wall-clock time since the beginning and then executed it to see whether or not it could bypass the WAF. We compare the performance of the techniques based on time scale and the cumulative number of bypassing tests generated over time.

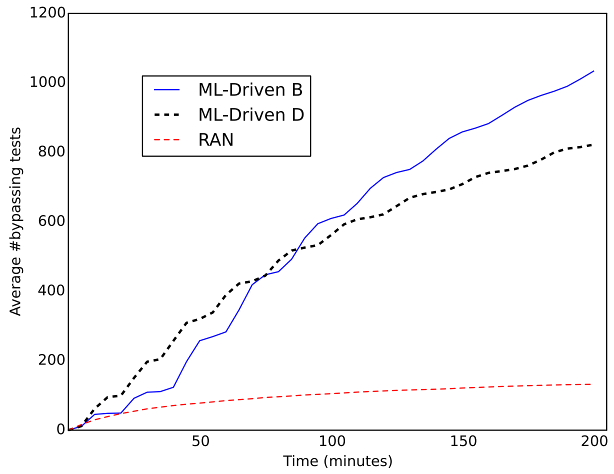


Fig. 6. Average of the performance on nine SOAP messages (10 repetitions each) for RAN, ML-Driven B, and ML-Driven D. The latter two were run with K = 40%.

To help us answer **RQ3**, Fig. 6 depicts the average number of distinct, bypassing tests generated over the amount of time required to obtain them. The data represents the average of 10 runs on the nine SOAP samples for each technique, measured within intervals of five minutes. A first observation is that all techniques can generate tests that bypass the WAF, meaning that the WAF is vulnerable to SQLi, putting online systems under its protection at risk. In fact, we observed that these bypassing tests can trigger SQLi attacks in HotelRS and SugarCRM. Second, the sharply increasing plots corresponding to ML-Driven B and ML-Driven D indicate that they are much more efficient than RAN, our baseline. ML-Driven D is at first more efficient but is then caught up by ML-Driven B, which ends up being more efficient in the long term. The plots are also slightly oscillating, thus depicting the effect of regular re-training of the classifier. The slopes of the plots reduce over time as it becomes increasingly harder to find new bypassing tests that have not yet been executed. Overall, the results show that the ML-Driven techniques outperform RAN by an order of magnitude regarding the number of distinct tests that are generated and can circumvent the WAF.

TABLE II
THE AVERAGE NUMBER OF FIREWALL-BYPASSING TESTS GENERATED BY RAN, ML-DRIVEN B & D AT DIFFERENT TIME FRAMES.

			TIME					
App.	**SOAP Sample**	**Tech.**	**5**	**10**		**60**	**90**	**180**
Cyclos	doPayment	ML.-B	10.3	39.3		245.0	505.4	940.4
	expireTicket	ML.-B	14.0	60.0		352.9	643.4	1,084.2
	simulatePayment	ML.-B	9.0	35.0		247.8	535.5	972.6
HotelRS	confirmRoom	ML.-B	10.1	44.3		321.1	572.4	1,007.8
	getCustomerByID	ML.-B	13.4	57.9		335.3	634.9	1,047.3
SugarCRM	get_entries	ML.-B	10.1	35.0		138.2	315.4	647.6
	get_relationships	ML.-B	14.5	61.2		351.1	653.3	1,055.3
	search_by_module	ML.-B	10.1	35.4		247.7	504.5	890.4
	set_entry	ML.-B	11.3	41.2		307.9	614.1	1,028.8
	AVERAGE		**11.4**	**45.5**		**283.0**	**553.2**	**963.8**
Cyclos	doPayment	ML-.D	10.7	53.3		377.2	532.3	791.1
	expireTicket	ML-.D	15.0	86.0		456.6	562.0	832.4
	simulatePayment	ML-.D	10.3	46.4		358.1	520.1	777.9
HotelRS	confirmRoom	ML-.D	11.4	60.8		392.8	556.7	827.4
	getCustomerByID	ML-.D	13.4	74.0		398.8	529.8	803.7
SugarCRM	get_entries	ML-.D	11.6	44.2		266.4	385.6	525.7
	get_relationships	ML-.D	14.2	65.7		434.3	570.6	867.8
	search_by_module	ML-.D	11.0	60.1		416.3	539.9	782.6
	set_entry	ML-.D	10.2	65.3		402.7	536.0	802.7
	AVERAGE		**12.0**	**61.8**		**389.2**	**525.9**	**779.0**
Cyclos	doPayment	RAN	9.6	21.6		76.0	90.4	115.0
	expireTicket	RAN	26.2	44.9		101.4	116.8	144.8
	simulatePayment	RAN	12.8	24.6		75.7	93.7	123.2
HotelRS	confirmRoom	RAN	14.7	29.0		82.8	103.6	149.2
	getCustomerByID	RAN	22.0	37.8		101.8	119.0	149.8
SugarCRM	get_entries	RAN	8.7	19.8		71.1	85.7	111.2
	get_relationships	RAN	20.8	34.9		96.6	115.9	142.9
	search_by_module	RAN	11.1	19.9		60.2	67.6	79.2
	set_entry	RAN	21.0	37.0		101.0	120.0	147.0
	AVERAGE		**16.2**	**29.9**		**83.3**	**98.7**	**124.8**

Table II provides a complementary view of the average number of bypassing tests generated by RAN, ML-Driven B, and ML-Driven D at different points in time. Initially, ML-Driven generates tests randomly until a data corpus is build. As a result, the performance of all techniques is similar. Once the data corpus is available, the ML-Driven techniques use machine learning to generate tests. As a consequence, their performance is improving. After 10 minutes, the ML-Driven techniques generate more bypassing tests compared to RAN with a factor of 2 to 3. The difference grows larger in favour of ML-Driven after that.

We measure the efficiency E of a technique to be the number of new and distinct bypassing tests the technique generates in a minute. Specifically, we calculate within every time slot of 20 minutes, how many new such tests are generated, and then normalise the outcome to a unit of one minute. Fig. 7 describes the change in efficiency of RAN, ML-Driven B, and ML-Driven D over time. It confirms that the efficiency of RAN decreases consistently over time, while the efficiency of the other two techniques increases in the first hour and then decreases. This is expected as machine learning improves our approach over time. But as the test space is increasingly covered, it is increasingly harder to find new tests, leading to a reduced efficiency over time. Nevertheless, the efficiency of ML-Driven techniques is always higher than that of RAN within the experiment duration.

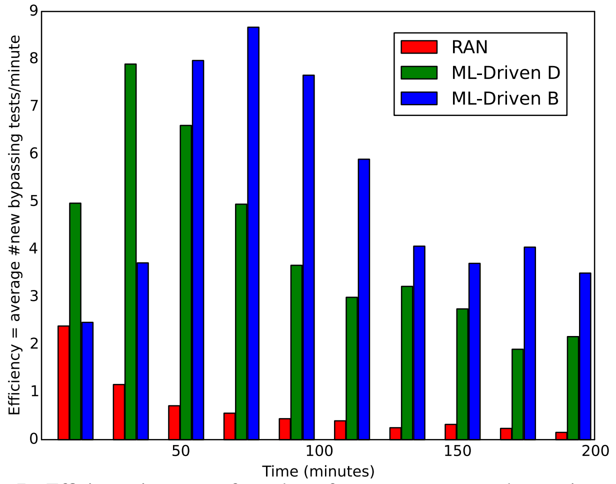


Fig. 7. Efficiency in terms of number of new tests generated per minute for RAN, ML-Driven B, and ML-Driven D

Comparing between ML-Driven B and ML-Driven D, the former favours alternative tests, while the later favours alternative slices of a fewer number of selected tests. Fig. 6 and 7 consistently show that ML-Driven D fares better than ML-Driven B at the beginning, but then its performance decreases faster than that of ML-Driven B. This is an interesting insight: during the course of test generation, it is first more efficient to start with few tests ranked higher than others by the classifier, trying more possible variants (by slice replacement). However, as time passes, in order to maintain the overall generation performance, we should select more tests and try a lesser number of their variants.

In addition to generating a large number of tests, which likely leads to revealing different holes in a target WAF, our ML-Driven techniques provide a security administrator with an invaluable decision tree and its path conditions, characterising successful attacks in terms of payload content. They can be useful in locating holes in the WAF. A decision tree learned through our iterative approach consists of nodes that represent relevant slices (small sets of characters, e.g., "/**/") determining, to some extent, whether a test (an attack string) bypasses the WAF. Furthermore, the path conditions representing the co-presence/absence of sets of slices provide additional information for debugging activities. For example, from the path condition $s_1 = 1 \wedge s_2 = 1 \wedge s_3 = 0$ of a bypassing test, we should narrow our search to the firewall rules that deal with the presence of s_1, s_2, and the absence of s_3 in order to fix them or to add new ones if needed. Investigating how to exploit the decision tree and path conditions in locating firewall holes is the subject of future work.

As all case studies, the generalisation of our findings is a threat to validity. To mitigate this threat, we have chosen the most popular open source WAF as subject, in combination with three applications, but further evaluations on other WAFs is necessary to improve generalisability.

V. Conclusion

Web application firewalls are an important layer to protect online systems from SQLi attacks. The increasing pace of new kinds of attacks and their sophistication requires the firewalls to be updated and tested regularly, as otherwise attacks can bypass them and get to the systems under protection. We present a machine learning-based testing approach to detect SQL injection vulnerabilities in firewalls. The approach automatically generates diverse attack payloads, which can be seeded into inputs of web-based applications, and then submit them to a system that is protected by a firewall. More importantly, by incrementally learning from the tests that are blocked or passed by the firewall, our approach can then select tests that exhibit characteristics associated with bypassing the firewall and mutate them to efficiently generate new successful attacks. Having such invaluable set of firewall-bypassing attacks, a security expert can fix or fine-tune the firewall rules so as to block imminent SQLi attacks. In the attacker-defender war, time is vital. Being able to learn and anticipate more attacks that can circumvent a firewall in a timely manner is very important in order to secure business data and services.

Our key contributions include a context-free grammar for SQLi attacks that can be used in various contexts (e.g., in testing web applications or services directly). Using the grammar, we devise an approach based on machine learning, ML-Driven, that can effectively and efficiently detect SQLi vulnerabilities in web application firewalls. The performance of ML-Driven is significantly higher than that of a random technique. Our experiments show that ML-Driven generates a large number of bypassing tests that pinpoint issues in the firewall. Having a large number of bypassing tests is essential for abstracting common patterns in successful attacks and to eventually fix the firewall.

As future work, we will investigate how the large number of bypassing attacks, which makes it infeasible to inspect them all manually, can be clustered into a manageable number of groups to help the firewall administrator identify the root cause for each group and fix the firewall's configuration. In addition, we will further evaluate our approach on different firewalls having different configurations.

Acknowledgements

This work was supported by the National Research Fund, Luxembourg (grant FNR/P10/03 and FNR4800382) and has been carried out with our industry collaborator, CETREL.

References

[1] E. Al-Shaer, A. El-Atawy, and T. Samak. Automated pseudo-live testing of firewall configuration enforcement. *Selected Areas in Communications, IEEE Journal on*, 27(3):302–314, 2009.

[2] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[3] N. Antunes, N. Laranjeiro, M. Vieira, and H. Madeira. Command injection vulnerability scanner for web services. http://eden.dei.uc.pt/ mvieira/.

[4] 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.

[5] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan. Automated testing for sql injection vulnerabilities: An input mutation approach. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 259–269, New York, NY, USA, 2014. ACM.

[6] W. Banzhaf, F. D. Francone, R. E. Keller, and P. Nordin. *Genetic Programming: An Introduction: on the Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

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

[8] A. D. Brucker, L. Brgger, P. Kearney, and B. Wolff. Verified firewall policy transformations for test case generation. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 345–354. IEEE, 2010.

[9] 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.

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

[11] W. 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, Arlington, VA, USA*, pages 13–15, 2006.

[12] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[13] T. K. Ho. The random subspace method for constructing decision forests. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(8):832–844, Aug 1998.

[14] J. Hwang, T. Xie, F. Chen, and A. X. Liu. Systematic structural testing of firewall policies. In *Reliable Distributed Systems, 2008. SRDS'08. IEEE Symposium on*, pages 105–114. IEEE, 2008.

[15] J. Jürjens and G. Wimmel. Specification-based testing of firewalls. In D. Bjørner, M. Broy, and A. Zamulin, editors, *Perspectives of System Informatics*, volume 2244 of *Lecture Notes in Computer Science*, pages 308–316. Springer Berlin Heidelberg, 2001.

[16] H. Liu and H. B. Kuan Tan. Testing input validation in web applications through automated model recovery. *Journal of Systems and Software*, 81(2):222–233, 2008.

[17] P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[18] J. Offutt, Y. Wu, X. Du, and H. Huang. Bypass testing of web applications. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 187–197. IEEE, 2004.

[19] D. Senn, D. Basin, and G. Caronni. Firewall conformance testing. In F. Khendek and R. Dssouli, editors, *Testing of Communicating Systems*, volume 3502 of *Lecture Notes in Computer Science*, pages 226–241. Springer Berlin Heidelberg, 2005.

[20] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos. Management of an Academic HPC Cluster: The UL Experience. In *Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014)*, pages 959–967, Bologna, Italy, July 2014. IEEE.