A Machine-Learning-Driven Evolutionary Approach for Testing Web Application Firewalls

Dennis Appelt, Cu D. Nguyen, Annibale Panichella, and Lionel C. Briand, *Fellow, IEEE*

***Abstract*—Web application firewalls (WAFs) are an essential protection mechanism for online software systems. Because of the relentless flow of new kinds of attacks as well as their increased sophistication, WAFs have to be updated and tested regularly to prevent attackers from easily circumventing them. In this paper, we focus on testing WAFs for SQL injection attacks, but the general principles and strategy we propose can be adapted to other contexts. We present `ML-Driven`, an approach based on machine learning and an evolutionary algorithm to automatically detect holes in WAFs that let SQL injection attacks bypass them. Initially, `ML-Driven` automatically generates a diverse set of attacks and submits them to the system being protected by the target WAF. Then, `ML-Driven` selects attacks that exhibit patterns (substrings) associated with bypassing the WAF and evolves them to generate new successful bypassing attacks. Machine learning is used to incrementally learn attack patterns from previously generated attacks according to their testing results, i.e., if they are blocked or bypass the WAF. We implemented `ML-Driven` in a tool and evaluated it on ModSecurity, a widely used open-source WAF, and a proprietary WAF protecting a financial institution. Our empirical results indicate that `ML-Driven` is effective and efficient at generating SQL injection attacks bypassing WAFs and identifying attack patterns.**

***Index Terms*—Software security testing, SQL injection (SQLi), web application firewall (WAF).**

I. Introduction

WEB application firewalls (WAFs) protect enterprise web systems from malicious attacks. As a facade to the web application they protect, WAFs inspect incoming HTTP messages and decide whether blocking or forwarding them to the target web application. The decision is often performed based on a set of rules, which are designed to detect attack patterns. Since cyber-attacks are increasingly sophisticated, WAF rules tend to become complex and difficult to manually maintain and test. Therefore, automated testing techniques for WAFs are crucial to prevent malicious requests from reaching web applications and services.

In this paper, we focus our testing efforts on a common category of attacks, namely, SQL injections (SQLi). SQLi has received a lot of attention from academia as well as practitioners [7], [10], [14], [24]–[27], [32], [34], [45]. Yet the Open Web Application Security Project (OWASP) finds that the prevalence of SQLi vulnerabilities is common and the impact of a successful exploitation is severe [50]. While we assess our approach based on the example of SQLi, we believe that many of the principles of our methodology can be adapted to other forms of attacks.

Various techniques have been proposed in the literature to detect SQLi attacks based on a variety of approaches, including white-box testing [21], static analysis [20], model-based testing [30], and black-box testing [18]. However, such techniques present some limitations that may adversely impact their practical applicability as well as their vulnerability detection capability. For example, white-box testing techniques and static analysis tools require access to source code [33], which might not be possible when dealing with third-party components or industrial appliances, and are linked to specific programming languages [19]. Model-based testing techniques require models expressing the security policies or the implementation of WAFs and the web application under test [30], which are often not available or very difficult to manually construct. Black-box testing strategies do not require models or the source code but they are less effective in detecting SQLi vulnerabilities. Indeed, comprehensive reviews on black-box techniques [13], [18] have revealed that many types of security vulnerabilities (including SQLi attacks) remain largely undetected and, thus, warrant further research.

In our preliminary work [9], we introduced a novel black-box technique, namely, `ML-Driven`, that combines the classical ($\mu + \lambda$) evolutionary algorithm (EAs) with machine learning algorithms for generating tests (i.e., attacks) bypassing a WAF's validation routines. `ML-Driven` uses machine learning to incrementally learn attack patterns and build a classifier, i.e., that predicts combinations of attack substrings ("slices") associated with bypassing the WAF. The resulting classifier is used within the main loop of ($\mu + \lambda$)-EAs to rank tests depending on which substrings compose them and the corresponding bypassing probabilities. In each iteration, tests with the highest rank are selected and mutated to generate λ new tests (offsprings), which are then executed against the WAF. The corresponding execution results are used to retrain the classifier to incrementally improve its accuracy. Through subsequent generations, tests are

Manuscript received August 1, 2017; revised January 12, 2018; accepted January 27, 2018. This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 Research and Innovation Programme under Grant 694277. Associate Editor: T. H. Tse. *(Corresponding author: Annibale Panichella.)*

The authors are with the SnT Centre, University of Luxembourg, Luxembourg City L-2721, Luxembourg (e-mail: dennis.appelt@googlemail.com; cdnguyen.dr@gmail.com; annibale.panichella@uni.lu; briand@svv.lu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2018.2805763

evolved to increase the number of attacks able to bypass the target WAF.

We defined two variants of `ML-Driven`, namely, `ML-Driven D` and `ML-Driven B`, that differ in the number of tests being selected for generating new tests (offsprings). The former variant selects fewer tests to evolve but generates more offsprings per selected test, thus increasing *exploitation* (deep search). The latter variant selects more tests to mutate, which results in a lower number of offsprings per selected test, hence increasing *exploration* (broad search). Our preliminary study with ModSecurity,[1] a popular open source WAF, has shown that both `ML-Driven D` and `ML-Driven B` are effective in generating a large number of distinct SQLi attacks passing thought the WAF. However, `ML-Driven D` performs better in the earlier stages of the search, while `ML-Driven B` outperforms it in the last part of the search, for reasons we will explain.

In the race against cyber-attacks, time is vital. Being able to learn and anticipate attacks that successfully bypass WAFs in a timely manner is critical. With more successful, distinct attacks being detected, the WAF administrator is in a better position to identify missed attack patterns and to devise patches that block all further attacks sharing the same patterns. Therefore, our goal is to devise a technique that can efficiently generate as many successful distinct attacks as possible. To this aim, in this paper, we extended our prior work and propose an adaptive variant of `ML-Driven`, namely, `ML-Driven E` (enhanced), that combines the strengths of `ML-Driven D` (deep search), and `ML-Driven B` (broad search) in an adaptive manner. In `ML-Driven E`, the number of offsprings is computed dynamically depending on the bypassing probability assigned to each selected test by the constructed classifier. Conversely, `ML-Driven` B and D generate a fixed number of offsprings per attack/test. Therefore, `ML-Driven E` is a more flexible approach that better balances eploration and exploitation over run time.

Moreover, we conducted a much larger empirical study with two popular WAFs that protect three open-source and 44 proprietary web services. The results show that the enhanced version of our technique (`ML-Driven E`) significantly outperforms the following:

1) its predecessors `ML-Driven B` and `ML-Driven D`;
2) a random test strategy, which serves as a baseline;
3) two state-of-the-art vulnerability detection tools, namely, `SqlMap` and `WAF Testing Framework`.

We also performed a qualitative analysis of the attacks generated by our approach and we found out that they enable the identification of attack patterns that are strongly associated with bypassing the WAFs, thus providing better support for improving the WAFs' rule set.

To summarize, the key contributions of this paper include the following.

1) Enhancing `ML-Driven` with an adaptive test selection and generation heuristic, which more effectively explores attack patterns with higher likelihood of bypassing the WAF. This enhanced `ML-Driven` variant is intended to replace its predecessors, leaving the practitioners with a single, yet the best, option.
2) Assessing the influence of the selected machine learning algorithm on the test results by comparing two alternative classification models, namely, RandomTree and RandomForest, which are both adapted to large numbers of features and datasets, but with complementary advantages and drawbacks.
3) Extending the previous evaluation and conducting a large-scale experiment on a proprietary WAF protecting a financial institution.
4) Comparing `ML-Driven` with `SqlMap` and `WAF Testing Framework`, which are state-of-the-art vulnerability detection tools.
5) A qualitative analysis showing that the additional distinct attacks found by `ML-Driven E` help security analysts design better patches compared to the other `ML-Driven` variants, RAN, and state-of-the-art vulnerability detection tools.

The remainder of this paper is structured as follows. Section II-A provides background information on WAFs as well as SQLi attacks and discusses related work. Section III details our approach followed by Section IV where we describe the design and procedure of our empirical evaluation. Section V describes the evaluation results and their implications regarding our research questions, while Section VI explains and illustrates how to use the generated attacks for repairing vulnerable WAFs. Further reflections on the results are provided in Section VII while Section VIII concludes this paper.

II. Background and Related Work

In this section, we provide background notions about SQLi vulnerabilities and describe existing white-box and black-box approaches aimed at uncovering them.

A. SQLi Vulnerabilities

In systems that use databases, such as web-based systems, the SQL statements accessing back-end databases are usually treated as strings. These strings are formed by concatenating different string fragments based on user choices or the application's control flow. Once an SQL statement is formed, it is submitted to the database server to be executed. For example, an 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.

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

SQLi is an attack technique in which attackers inject malicious SQL code fragments into input parameters that lack proper validation or sanitization. An attacker might construct input values in a way that changes the behavior of the resulting SQL statement and performs arbitrary actions on the database (e.g., exposure of sensitive data, insertion or alteration of data without authorization, 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 --`, then 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 hand 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 blacklist. A blacklist contains string patterns, typically defined as regular expressions. Requests recognized 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 SQLi, one of the main types of attacks 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 [38]. Tests are generated to intentionally violate the client-side input checks and are then sent to the server application to test whether the input constraints are adequately evaluated. Liu and Kuan Tan [35] 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. Desmet *et al.* [17] verify a given combination of a WAF and a web application for broken access control vulnerabilities, e.g., forceful browsing, by explicitly specifying the interactions of application components on the source code level and by applying static and dynamic verification to enforce only legal state transitions. 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 recognized by the firewall as SQLi attacks and generate bypassing test cases that avoid those patterns.

Tripp *et al.* [48] proposed *XSS Analyzer*, a learning approach to web security testing. The authors tackle the problem of efficiently selecting from a comprehensive input space of attacks by learning from previous attack executions. Based on the performed learning, the selection of new attacks is adjusted to select attacks with a high probability of revealing a vulnerability. More specifically, *XSS Analyzer* generates attacks from a grammar and learns constraints that express which literals an attack cannot contain in order to evade detection. The authors find that *XSS Analyzer* outperforms a comparable state-of-the-art algorithm, thus, suggesting that learning input constraints (a concept similar to the path conditions in `ML-Driven`) is effective to guide the test case generation.

In contrast to our work, *XSS Analyzer* applies learning to individual literals only while our approach also learns if a combination of literals is likely to bypass or be blocked. Therefore, *XSS Analyzer* cannot capture more complex input constraints involving multiple literals simultaneously, e.g., an attack should contain literal *a*, but not *b* and *c* in order to evade detection. Furthermore, to analyze which literals in an attack are blocked, *XSS Analyzer* first splits each attack into its composing tokens; then, it resends each token to the target web application. Since multiple attacks can share the same tokens, *XSS Analyzer* sends each token only once to avoid performing the same analysis multiple times.

This procedure consumes a large quantity of HTTP requests. In contrast, `ML-Driven` does not require to resubmit individual slices, but learns path conditions solely from the previously executed test case and, thus, spends its test budget more efficiently. In addition, there are differences between *XSS Analyser* and `ML-Driven` in terms of objectives: The former addresses cross-site scripting sanitization in web applications, while the latter addresses the detection of SQLi attacks in WAFs.

In grammar-based testing, a strategy typically samples a large input space defined by a grammar. Several grammar-based approaches exist in the literature for testing security properties of an application under test [37], [47]. Godefroid *et al.* [23] proposed white-box fuzzing, which starts by executing an application under test with a given well-formed input and uses symbolic execution to create input constraints when conditional statements are encountered on the execution path. Then, new inputs are created that exercise different execution paths by negating the previously collected constraints. The implementation of the approach found a critical memory corruption vulnerability in a file-processing application. In a follow-up work, the authors propose grammar-based white-box fuzzing [21], which addresses the generation of highly structured program inputs.

In this paper, well-formed inputs are generated from a grammar and the constraints created during execution are expressed as constraints on grammar tokens. To generate new inputs, a constraint solver searches the grammar for inputs that satisfy the constraints. The authors implemented their work in a tool named *SAGE* and found several security-related bugs [22]. In contrast to the mentioned work of Godefroid *et al.*, our work does not require access to the source code of the application under test, which is in many practical scenarios not available, but proposes a black-box approach based on machine learning to efficiently sample the input space defined by an attack grammar.

The topic of testing network firewalls has also been 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 WAF policies. Brucker *et al.* [1] proposed a model-based testing approach which transforms a firewall policy into a normal form. 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.* [29] defined structural coverage criteria of policies under test and developed a test generation technique based on constraint solving that tries to maximize structural coverage. Other research has focused on testing the firewalls implementation instead of policies. Al-Shaer *et al.* [3] developed a framework to automatically test if a policy is correctly enforced by a firewall. 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 and Wimmel [30] proposed to formally model the tested firewall and to automatically derive test cases from the formal specification. Senn *et al.* [43] proposed a formal language for specifying security policies and automatically generate test cases from formal policies to test the firewall. 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

This section introduces an approach for testing WAFs. Section III-A defines the input space for this testing problem as a context-free grammar. Section III-B presents a simple attack generation strategy that randomly samples the input space and serves as baseline. Section III-C presents two test generation strategies that make use of machine learning to guide test generation toward areas in the input space that are more likely to contain successful attacks. Finally, Section III-D details an approach to combine such attack generation strategies to achieve better results.

A. Context-Free Grammar for SQLi Attacks

SQLi attacks (or test cases in this context) are small "programs" that aim at changing the intent of the target SQL queries they are injected in. We systematically surveyed known SQLi attacks published in the literature, e.g., [5], [6], [24], and from other sources, e.g., OWASP[2] and SqlMap.[3] Then, we defined a context-free grammar for generating and analyzing SQLi attacks.

We consider three main categories of SQLi attacks in our grammar: 1) *Boolean*; 2) *Union*; and 3) *Piggy-backed*. These type of attacks aim at manipulating the intended logic by injecting additional SQL code fragments in the original SQL queries. We briefly discuss each attack category and provide example attacks that can be derived using our grammar. For a detailed discussion, refer to the literature [10], [27].

1) Boolean Attacks: The intent of a *Boolean SQLi attack* is to influence the *where clause* within an SQL statement to always evaluate either to true or false. As a result, a statement, into which a *Boolean SQLi attack* is injected, returns on its execution either all data records of the queried database tables (in case the where clause evaluates always to true) or none (in case the where clause evaluates always to false). This attack method is typically used to bypass authentication mechanisms, extract data without authorization, or to identify injectable parameters. The example described in Section II-A is an instance of Boolean attacks.

2) Union Attacks: The `union` keyword joins the result set of multiple `select` statements and, hence, *union SQLi attacks* are typically used to extract data located in other database tables than the original statement is querying. For example, consider an application that retrieves a list of product names based on a search term. The SQL statement to retrieve the product names might be

```
    SELECT name FROM products WHERE name
LIKE "%search term%"
```

where `search term` is a string provided by the user. If the SQL statement is formed in an insecure way, an attacker could provide the search term `phone%" UNION SELECT passwd FROM users #%"`, which would result in the statement

```
    SELECT name FROM products WHERE name
LIKE "%phone%" UNION SELECT passwd FROM
users.
```

Hence, in addition to a list of products containing the search term `phone`, the attacker could obtain the passwords of all users with the modified query above.

3) Piggy-Backed Attacks: In SQL, the semicolon (;) can be used to separate two SQL statements. *Piggy-backed attacks* use the semicolon to append an additional statement to the original statement and can be used for a wide range of attack purposes (e.g., data extracting or modification, and denial of service). An example of a piggy-backed attack is `; DROP TABLE users #`. If this attack is injected into a vulnerable SQL statement, it drops the table `user` and, thus, potentially breaks the application.

[2] [Online]. Available: https://www.owasp.org

[3] [Online]. Available: http://sqlmap.org

```
1. ⟨start⟩ ::= ⟨numericCtx⟩ | ⟨sQuoteCtx⟩ | ⟨dQuoteCtx⟩ ;

Injection Context
2. ⟨numericCtx⟩ ::= ⟨digitZero⟩, ⟨wsp⟩, ⟨booleanAtk⟩, ⟨wsp⟩
     |  ⟨digitZero⟩, ⟨parC⟩, ⟨wsp⟩, ⟨booleanAtk⟩, ⟨wsp⟩, ⟨operOr⟩,
        ⟨parO⟩, ⟨digitZero⟩
     |  ⟨digitZero⟩, [⟨parC⟩], ⟨wsp⟩, ⟨sqliAtk⟩, ⟨comment⟩ ;
3. ⟨sQuoteCtx⟩ ::= ⟨squote⟩, ⟨wsp⟩, ⟨booleanAtk⟩, ⟨wsp⟩, ⟨operOr⟩,
        ⟨squote⟩
     |  ⟨squote⟩, ⟨parC⟩, ⟨wsp⟩, ⟨booleanAtk⟩, ⟨wsp⟩, ⟨operOr⟩,
        ⟨parO⟩, ⟨squote⟩
     |  ⟨squote⟩, [⟨parC⟩], ⟨wsp⟩, ⟨sqliAtk⟩, ⟨comment⟩ ;
4. ⟨dQuoteCtx⟩ ::= ⟨dquote⟩, ⟨wsp⟩, ⟨booleanAtk⟩, ⟨wsp⟩, ⟨operOr⟩,
        ⟨dquote⟩
     |  ⟨dquote⟩, ⟨parC⟩, ⟨wsp⟩, ⟨booleanAtk⟩, ⟨wsp⟩, ⟨operOr⟩,
        ⟨parO⟩, ⟨dquote⟩
     |  ⟨dquote⟩, [⟨parC⟩], ⟨wsp⟩, ⟨sqliAtk⟩, ⟨comment⟩ ;
5. ⟨sqliAtk⟩ ::= ⟨unionAtk⟩ | ⟨piggyAtk⟩ | ⟨booleanAtk⟩ ;

Union Attacks
6. ⟨unionAtk⟩ ::= ⟨union⟩, ⟨wsp⟩, [⟨unionPostfix⟩], ⟨operSel⟩, ⟨wsp⟩,
        ⟨cols⟩
     |  ⟨union⟩, ⟨wsp⟩, [⟨unionPostfix⟩] , ⟨parO⟩, ⟨operSel⟩, ⟨wsp⟩,
        ⟨cols⟩,⟨parC⟩ ;
7. ⟨union⟩ ::= ⟨operUni⟩ | "/*!", ["50000"], ⟨operUni⟩, "*/" | ... ;
8. ⟨unionPostfix⟩ ::= "all", ⟨wsp⟩ | "distinct", ⟨wsp⟩;

Piggy-Backed Attacks
9. ⟨piggyAtk⟩ ::= ⟨operSem⟩, ⟨operSel⟩, ⟨wsp⟩, ⟨funcSleep⟩ | ... ;

Boolean Attacks
10. ⟨booleanAtk⟩ ::= ⟨orAtk⟩ | ⟨andAtk⟩ ;
11. ⟨orAtk⟩ ::= ⟨operOr⟩, ⟨booleanTrueExpression⟩ ;
12. ⟨andAtk⟩ ::= ⟨operAnd⟩, ⟨booleanFalseExpression⟩ ;
13. ⟨booleanTrueExpression⟩ ::= ⟨unaryTrue⟩ | ⟨binaryTrue⟩ ;

SQL Operators and Keyword
15. ⟨operNot⟩ ::= "!" | "not" ;
16. ⟨operBinInvert⟩ ::= " " ;
17. ⟨operEqual⟩ ::= "=" ;
18. ⟨operLt⟩ ::= "<" ;
19. ⟨operGt⟩ ::= ">" ;
20. ⟨operLike⟩ ::= "like" ;
21. ⟨operIs⟩ ::= "is" ;
22. ⟨operMinus⟩ ::= "-" ;
23. ⟨operOr⟩ ::= "or" | "||" ;
24. ⟨operAnd⟩ ::= "and" | "&&" ;
25. ⟨operSel⟩ ::= "select"
26. ⟨operUni⟩ ::= "union" ;
27. ⟨operSem⟩ ::= ";" ;
28. ⟨comment⟩ ::= "#" | ⟨ddash⟩, ⟨blank⟩;
29. ⟨ddash⟩ ::= "−−"

Obfuscation
30. ⟨inlineComment⟩ ::= "/**/" ;
31. ⟨blank⟩ ::= "␣" ;
32. ⟨wsp⟩ ::= ⟨blank⟩ | ⟨inlineComment⟩
```

Fig. 1. Excerpt of the grammar used in this paper expressed in extended Backus normal form.

We defined a grammar for SQLi attacks in the extended Backus normal form, which is publicly available[4] on GitHub. An excerpt of the grammar is depicted in Fig. 1. The start symbol of the grammar is ⟨*start*⟩, while "::=" denotes the production symbol, "," is concatenation, and "|" represents alternatives (grammar rule 1 of Fig. 1). The grammar covers three different contexts in which SQLi attacks can be inserted into: *numericCtx*, *sQuoteCtx*, and *dQuoteCtx*. SQLi attacks belonging to the first context yield syntactically correct statements if they are injected in a numerical context (rule 2 of Fig. 1). For example, the attack `1 OR true` belongs to *numericCtx* and yields a syntactically correct statement if injected into the statement `SELECT * FROM person WHERE age=`< *user_input*> where the placeholder < *user_input*> is intended to be replaced with a numerical value. Similarly, the second and third context, *sQuoteCtx* and *dQuoteCtx*, target SQL statements in which the user input is used as string literal and surrounded by single quotes (rule 3 of Fig. 1) and double quotes (rule 4), respectively. For example, the SQLi attack `"OR"a"="a` belongs to *dQuoteCtx* and yields a syntactically correct statement if injected into `SELECT * FROM person where name="`< *user_input*>`"`.

Our grammar can be extended to incorporate other variants of SQLi attacks. For example, the nonterminal ⟨*blank*⟩ (rule 30 of Fig. 1) 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. Since the grammar is an input to our approach, it can also be replaced with alternative grammars, which define an input space for different attack types (e.g., cross-site scripting or XML injection).

B. Grammar-Based Random Attack Generation (RAN)

Based on the proposed grammar, the random attack generation procedure is straightforward: Beginning from the start symbol ⟨*start*⟩, a randomly selected production rule is applied recursively until only terminals are left. Since there is no loop in the grammar, this attack generation procedure will always terminate. The output SQLi attack is produced by concatenating all terminal symbols appearing in the generated instance of the grammar.

To produce a set of diverse random SQLi attacks that yield a good coverage of the grammar, each production rule is selected with a probability proportional to the number of distinct production rules descending from the current one.

Among the techniques presented in this paper, RAN implements the simplest strategy for sampling the input space defined by the attack grammar. Indeed, multiple SQLi attacks can be generated by simply reapplying RAN multiple times until the maximum number of tests/attacks (or the maximum running time) is reached. Bypassing attacks are then detected by executing all generated SQLi attacks against the target WAF.

C. Machine Learning-Guided Attack Generation

As the difficulty to find bypassing attacks increases, i.e., a WAF detects a large proportion of attacks, an RAN strategy will increasingly become inefficient. In such situations, a more advanced test generation strategy that spends more computational effort to identify test cases with a higher bypassing probability is expected to be more efficient. In this section, we introduce a machine learning-based approach, called `ML-Driven`, to sample the input space in a more efficient manner than RAN does.

Theoretically, SQLi attacks could be generated using search-based testing techniques [4], [12], [36], [40]. However, the application of these techniques requires the definition of a *distance*

[4] https://github.com/dappelt/xavier-grammar

function f that measures the distance between a coverage target to reach and the execution results of a given test case [36], [39], [46]. For example, in white-box unit testing, *approach level* [36] and *branch distance* [36] are used to determine how far is the execution path of test t from covering a given branch in the control flow graph. However, in the context of security testing, the target SQLi vulnerabilities are not known *a priori* and, thus, such a distance function f cannot be defined. Therefore, 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 test. The problem is challenging because there is little information available to estimate 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. Lack of guidance is a well-known issue to address when applying search-based techniques since previous studies (e.g., [44]) showed that sophisticated search algorithms (e.g., EA) do not provide any advantage over random search (i.e., the approach in Section III-B) when the gradient of the function f has many plateaus (*flag problem*).

To tackle this problem, we introduce `ML-Driven`, a search-based technique that uses machine learning to model how the elements (attributes of attacks) of the tests are associated with the likelihood of bypassing the WAF, the latter being used to guide the search. In the search process, tests that are predicted to have such high likelihood are considered to have a high fitness and are more likely to be generated. `ML-Driven` first employs random test generation, as described in Section III-B, to generate an initial training set. Then, these tests are sent to a web application protected by the WAF, and are labelled as “P” or “B” depending on whether they bypass or are blocked by the WAF, respectively. Tests and execution results are then encoded and used as initial training data to learn a model estimating the likelihood (f) with which tests can bypass the WAF. Using this measure, we can rank, select, and modify 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 the procedure used to decompose and encode test cases into a training set, and the machine learning algorithms that are used to estimate the likelihood of each test to bypass the WAF. Finally, we describe `ML-Driven`, our search-based approach guided by machine learning.

1) Attack Decomposition: We can derive a test from the grammar by applying recursively its production rules. This procedure can be represented as a derivation tree. 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 represents a nonterminal symbol, a leaf node represents a terminal symbol, and an edge represents the applied production. Fig. 2 depicts the derivation tree of the Boolean attack: `'␣OR"a"="a"#`. In the course of generating this test, we first apply the $\langle start \rangle$ rule

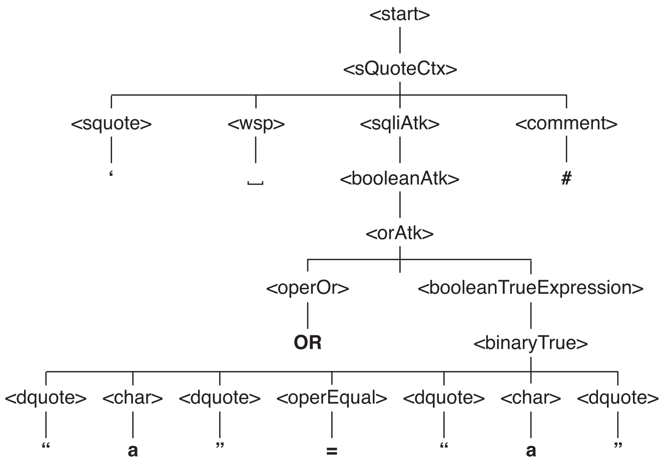

Fig. 2. Derivation tree of the “Boolean” SQLi attack: `'␣OR"a"="a"#`.

$$<start> ::= \langle \text{numericCtx} \rangle \mid \langle \text{sQuoteCtx} \rangle \mid \langle \text{dQuoteCtx} \rangle$$

and derive $\langle sQuoteCtx \rangle$. We then apply the third rule of the grammar to derive $\langle squote \rangle$, $\langle wsp \rangle$, $\langle sqliAtk \rangle$, and $\langle comment \rangle$. This procedure is repeated until all nonterminal symbols are transformed into terminal symbols. The attack string represented by a derivation tree is obtained by concatenating the leaves from left to right.

We use derivation trees to identify which substrings of an SQLi attack are likely to be responsible for the attack being blocked or not. Specifically, an attack is divided into substrings by decomposing its derivation tree into slices. The definition of a slice is as follows.

Definition 1: (Slice). A slice s is a subtree T' of a derivation tree T such that T' contains a strict subset of leaves of T.

A slice is supposed to represent a substring of an attack, hence only subtrees of a derivation tree that contain a subset of leaves are considered to be slices. Otherwise, if the subtree contains the same leaves as the derivation tree, the represented string is not a substring, but the same string as the derivation tree. For example, for the derivation tree in Fig. 2, the subtree with the root $\langle sQuoteCtx \rangle$ contains all the leaves of the whole derivation tree and, therefore, is not a slice. A sample of four slices decomposed from the derivation tree of this example is depicted in Fig. 3. Among all possible slices of a derivation tree, we are interested in extracting the minimal ones, i.e., slices that cannot be further divided into subslices.

Definition 2: (Minimal Slice). A slice s is minimal if it has only two nodes: a root and only one child that is a leaf.

The procedure to decompose a derivation tree into slices is detailed in Algorithm 1. Starting from the root node, the algorithm recursively computes slices from descendant nodes by calling *VISIT(child, root, S)* in line 5. In line 11, the condition $(root.leafs \setminus node.leafs) \neq \emptyset$ ensures that only subtrees complying with Definition 1 are considered. In line 14, the recursion ends if the node forms a minimal slice, as defined in Definition 2, otherwise the recursion continues. For example,

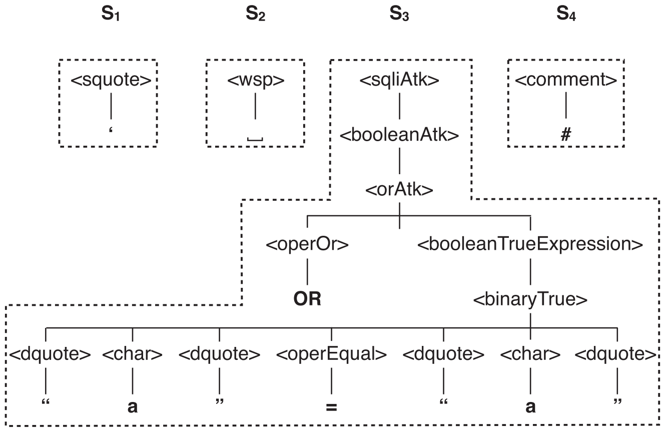

Fig. 3. Example subset of slices decomposed from the tree in Fig. 2.

TABLE I
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

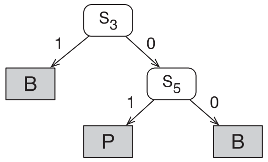

Fig. 4. Example of a DT obtained from the training data in Table I.

Algorithm 1: Tree Decomposition into Slices.

```
procedure DECOMPOSETREE($root$)
    $S \leftarrow \emptyset$
    $children \leftarrow root.childNodes$
    for all $child \in children$ do
        VISIT($child, root, S$)
    end for
    return $S$
end procedure
procedure Visit ($node, root, S$)
    $s \leftarrow getSlice(node)$        ▷ get the slice for which node is the root
    if $(root.leafs \setminus node.leafs) \neq \emptyset$ then
        $S \leftarrow S \cup s$
    end if
    if s is minimal then
        return
    else
        $children \leftarrow node.childNodes$
        for all $child \in children$ do
            VISIT($child, root, S$)
        end for
    end if
end procedure
```

applying the decomposition procedure to the derivation tree in Fig. 2 yields a set of 12 distinct slices.

We conjecture that the appearance of one or more slices in a test could result in the test getting blocked or not. In the following sections, we develop this idea further by analyzing slices of a collection of tests and predicting, using machine learning, how their appearance in the tests affects their likelihood of bypassing a WAF or being blocked.

2) Training Set Preparation: Given a set of tests that have been labeled with their execution result against 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 attack decomposition procedure.
2) Each slice is assigned a globally unique identifier. If the same slice is a part of multiple tests, it is referenced by the same identifier. We map each unique slice to an attribute (a feature) of the training dataset for machine learning.
3) Every test is transformed into an observation of the training dataset 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, let us consider 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 representation on the right side of the table. In total, we have five unique slices from all the tests and they become attributes of the training data for machine learning. If a slice appears in a test, its corresponding attribute value in the training data is “1,” and otherwise “0.”

3) Decision Tree (DT) and Path Condition: By decomposing tests into slices and transforming them into a labeled dataset, we can now apply a supervised machine learning technique to predict which slices or combinations of slices are associated with tests bypassing or being blocked by the WAF. To be able to identify such slices, we rely on machine learning techniques that provide an interpretable output model. That is, the reason for the classification of attacks into bypassing or blocked should be easily comprehensible. Therefore, though other algorithms generating classification rules could have been used as well, we selected DTs for this task.

A DT is a tree structure with decision nodes and leaves; a decision node represents an attribute from a dataset; each branch from such a node represents a possible value for the corresponding attribute; and a leaf node represents a classification for all instances that reach this node. In our context, each decision node represents a slice and the branches from the node can be “0” or “1,” corresponding to whether the slice is absent or present. A leaf node classifies instances into blocked or bypassing and is labeled accordingly with “B” or “P.” Fig. 4 shows an example DT learned from the data in Table I.

The paths from the root node of the DT to its leaf nodes embody the combinations of slices which are likely to be the reason for tests to bypass or to be blocked, depending on the classification. More generally, we define a concept of *path condition* as follows.

Definition 3: (Path Condition) A path condition represents a set of slices that the machine learning technique deems to be relevant for the attack's classification into blocked or bypassing. The path condition is represented as a conjunction $\bigwedge_i^k (s_i = val)$*, in which* $val = 1 \mid 0$*, and* k *is the number of relevant slices.*

The procedure for computing path conditions depends on the machine learning algorithm that is used to build DTs. We have selected two alternative algorithms and assessed their overall impact on test results.

RandomTree: The most prominent difference to classical DT algorithms (e.g., C4.5 [42]) is that RandomTree relies on randomization for building the DT [15]. When selecting an attribute for a tree node, the algorithm chooses the best attribute amongst a randomly selected subset of attributes. By choosing only subset of attributes, the algorithm scales well with the size of the training dataset. In our context, a scalable learner is important since the datasets contain a larger number of attributes and the DT is frequently rebuilt, as described in Section III-C4.

Given a DT and a slice vector V of a test t, we can obtain the path condition for t by visiting the DT from the root and check the presence (value $= 1$) or absence (value $= 0$) of the attributes encountered with respect to V. The procedure stops once a leaf is reached. For instance, Fig. 4 presents a DT learned from the example data discussed in Table I. For test t_1, the attribute s_3 is present in the test's slice vector $\langle s_1, s_2, s_3 \rangle$, and thus, t_1 follows the left branch. Hence, the path condition is $(s_3 = 1)$. Similarly, for test t_3 with the slice vector $\langle s_4, s_5 \rangle$, the attribute s_3 is not present, thus the right branch is followed leading to attribute s_5, which is present in the slice vector. Therefore, the resulting path condition for t_3 is $(s_3 = 0 \wedge s_5 = 1)$.

RandomForest: Machine classification methods are well known to be unstable, i.e., a small change in the training data can result in a different classification model [51]. Ensemble methods have been proposed to address the issue. In essence, multiple models are learned so that their collective predictions can mitigate bias in individual models. In this paper, we implement an ensemble of classifiers to guide the test generation, i.e., instead of using only one RandomTree, we extend our technique to make use of ensembles of trees produced by RandomForest [15]. However, the benefits of RandomForest come at the cost of an increased computational overhead, e.g., learning an ensemble of RandomTrees takes longer than learning only a single RandomTree. We evaluate in Section V whether the benefits justify the increased overhead.

A RandomForest consists of multiple RandomTrees. To classify an attack with RandomForest, each individual RandomTree first classifies the attack and computes the prediction confidence (an estimated probability for the classification is to be correct). Then, all individual classifications are consolidated by computing the average of the prediction confidence values for each class. Finally, the class with the highest prediction confidence is chosen by RandomForest as final classification.

Algorithm 2: ML-driven SQLi Attack Generation.

```
1:  procedure MLDRIVENGEN($initTests$, $outputTests$)
2:      $execute(initTests)$
3:      $P \leftarrow initTests$
4:      $archive \leftarrow$ UPDATEARCHIVE($P$)
5:      // learn the initial classifier
6:      $trainData \leftarrow transform(archive)$
7:      $DT \leftarrow learnClassifier(trainData)$
8:      $rankTests(P, DT)$
9:      while $not\text{-}done$ do
10:         O $\leftarrow$ OFFSPRINGSGEN($P$, λ, MAX$_M$)
11:         $execute(O)$
12:         $archive \leftarrow$ UPDATEARCHIVE($O$)
13:         // re-training the classifier
14:         $trainData \leftarrow transform(archive)$
15:         $DT \leftarrow learnClassifier(trainData)$
16:         // new population
17:         $P \leftarrow$ SELECT($P \cup O$)
18:     end while
19:     $outputTests \leftarrow filterBypassingTests(archive)$
20:     return $outputTests$
21: end procedure
```

To compute the path condition for a given attack with RandomForest, for all trees that classify the attack as bypassing, a path condition is computed separately. Thereby, the path condition for each tree is computed according to the previously described procedure. The overall path condition for the entire RandomForest is, then, the conjunction of the path conditions computed from the trees.

4) ML-Driven Evolutionary Testing Strategy: To increase the likelihood of generating bypassing attacks, we propose an ML-driven evolutionary testing strategy whose pseudocode is detailed in Algorithm 2. It combines machine learning algorithms (either *RandomTree* or *RandomForest*) with the classical (μ + λ) EA, which is a population-based EA with a population size μ and a restricted number of λ offsprings.

As any EA, Algorithm 2 starts with an initial set of solutions ($initTests$) usually called *population*. The initial population can either be generated by the random attack generator from Section III-B or by selecting tests from existing test suites. The initial tests are executed against a target WAF (line 2) if their execution results are not yet known (i.e., in case of randomly generated tests). Each solution is assigned a fitness score according to its probability of bypassing the WAF determined the classifier built in lines 5 and 6 of Algorithm 2. The classifier is built as follows: 1) the tests are transformed into the training set (routine $trainData$ in line 6); 2) a classifier DT is then trained from the data (line 7 of Algorithm 2). Then, individuals in the initial population are ranked using the DT classifier (routine $rankTests$ in line 8), which assigns each individual (test) a probability of bypassing the WAF.

In the main loop in lines 9–18, the population is evolved through subsequent iterations, called *generations*. λ new solutions are generated using the routine OffspringsGen in line 10. Such a routine selects the best individuals (parents) and mutates them to generate λ new individuals, called *offsprings*. The offsprings are executed against the target WAF (line 11) and labeled as "bypassing" or "blocked" depending on their execution results. In line 12, the newly generated tests are added to an *archive* (line 12), which is a second population used to keep track of all tests being generated across the generations [39]. Then, the DT classifier is retrained by converting the archive into the training set (line 14) and reapplying the ML training (line 15). At the end of each generation, a new population is formed in line 17 by selecting the fittest μ tests among parents and offsprings according the DT classifier. The loop terminates when the maximum number of generations (or the maximum running time) is reached (condition *not-done* in line 9).

Therefore, the three key components of Algorithm 2 are as follows:

1) the ML-driven fitness assignment;

2) the selection operator; and

3) the mutation operator applied for generating offsprings.

These key components are detailed in the following paragraphs.

ML-driven fitness assignment: In this paper, each candidate test t is assigned a fitness score by using machine learning algorithms. In Algorithm 2, a classifier DT is built at the beginning of the search (lines 6 and 7) at the end of each generation (lines 14 and 15 of Algorithm 2) using the *archive* as training set. Such an *archive* is updated whenever new test cases (either "bypassing" or "blocked") are generated (lines 4 and 12). To build the classifier, the archive is transformed into a training set (lines 6 and 7) using the steps discussed in Section III-C2. Then, the classifier DT is trained using either *RandomTree* or *RandomForest*. Once built, DT can be used to estimate the probability of a given test t to bypass the target WAF depending on which slices appear in the three decomposition of t (see Section III-C1).

Elitist selection: In (μ + λ)-EAs, μ parents and λ offsprings compete to form the new population of μ individuals for the next generation. In Algorithm 2, this procedure is implemented by the routine Select (line 17). Once the fitness assignment has been performed using DT, the tests in the current population are ranked in descending order of their bypassing probability. The top μ test cases in the ranking are selected to form the next population.

Generating offsprings: The routine used to generate offsprings is detailed in Algorithm 3. Given a set of μ parents, such a routine generates mutants from each parent until reaching a total number of λ offsprings (loop condition in line 3 of Algorithm 3). In each iteration of the loop in lines 3–15, Algorithm 3 selects the test with the highest rank as candidate test for mutation. If more than one candidate have tied ranks, the selection is random among them. If a test has been selected before, it will not be selected again. For each selected parent t (line 4), offsprings are generated starting from its corresponding path condition ($pathCondition$), which is determined using the routine $getPath$ (line 6) from the slice vector V of the attack t (obtained in line 5).

Algorithm 3: Offsprings Generation.

```
 1: procedure OffspringsGen($parents$, λ, MAX$_M$)
 2:   offsprings ← ∅
 3:   while |offsprings| < λ do
 4:     $t \leftarrow selectTest(parents)$
 5:     $V \leftarrow getSliceVector(t)$
 6:     $pathCondition \leftarrow getPath(V, DT)$
 7:     $s \leftarrow pickASliceFrom(V)$
 8:     while $s \neq null$ do
 9:       if $satisfy(s, pathCondition)$ then
10:         $newTests \leftarrow mutate(t, s,$ MAX$_M)$
11:         offsprings ← offsprings ∪ $newTests$
12:       end if
13:       $s \leftarrow pickASliceFrom(V)$
14:     end while
15:   end while
16:   return offsprings
17: end procedure
```

Then, offsprings are generated from a parent t by replacing its slices s in V with other alternative slices according to our grammar. In particular, one slice s is randomly picked from the slice vector V of the attack t (line 7) and it is replaced with an alternative slice s' to generate new tests (routine $mutate$ in line 10). Both s and s' have to satisfy the given path condition. That is, they either appear in the predicate and comply with it, or do not appear in the predicate of the path condition. If s does not satisfy the path condition, it is not considered for generating mutants/offsprings (condition in line 9).

To better explain this mutation procedure, let us consider the test t_2 from Table I with slice vector $\langle s_1, s_2, s_4\rangle$. According to the slices in Fig. 3, the path condition for t_2 is $(s_1 = 1)$; thus, 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 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

<*wsp*> ::= ⟨blank⟩ | ⟨inlineComment⟩.

As a result, we determine only one alternative slice that starts with ⟨*wsp*⟩ and derives ⟨*inlineComment*⟩.

At the mutation step in line 10 of Algorithm 3, the parameter MAX$_M$ is an integer value that limits the number of mutants that are generated for test t and slice s. If the number of alternative slices for t and s is greater than MAX$_M$, only MAX$_M$ alternative slices are selected and used, in turn, to form mutants. If the number of alternative slices is lower than MAX$_M$, all available slices are used to form mutants. Therefore, at most MAX$_N$ offsprings (mutants) are generated from each parent t, where each offspring differs from its parent in one single (alternative) slice.

Motivations: Let us now explain why we decided to opt for the classical (μ, λ)-EA, which is a mutation-based EA with no *crossover* operator. With this algorithm, MAX_N offsprings are generated from one single test t via mutation only. Crossover, which is another well-known operator widely applied in various EAs (e.g., genetic algorithms), is not applied here. Usually, it generates two offsprings from a pair of solutions (parents) by randomly exchanging their genes. However, for our problem, a crossover operator cannot be defined since different solutions (tests) within the same population have different derivation trees that represent instances of incompatible derivation rules of our grammar. For example, let us assume we select two test cases t_1 and t_2 for reproduction. The former instantiates the derivation rule ⟨*start*⟩ ::= ⟨*numericCtx*⟩ while the latter is an instance of the rule ⟨*start*⟩ ::= ⟨*sQuoteCtx*⟩. If we apply any crossover operator (e.g., the single-point crossover), then the resulting two offsprings would violate our grammar since each of the new test case would contain slices from different (incompatible) derivation rules.

Exploration Versus Exploitation: Assuming that the total number of offsprings (λ) to generate is constant, MAX_M controls how the EA explores the test space either *broadly* (exploration) or *deeply* (exploitation). When MAX_M is small, the approach generates fewer offsprings per selected test, but selects more tests to mutate, thus exploring the test space in a broader fashion (higher exploration). When MAX_M is large, on the contrary, the approach generates more offsprings per selected test, but selects fewer tests to mutate, thus exploring the test space in a deeper fashion (higher exploitation). In the evaluation section, two variants of `ML-Driven` are distinguished: `ML-Driven B` (broad, with $\text{MAX}_M = 10$) and `ML-Driven D` (deep, with $\text{MAX}_M = 100$). Section III-D presents a thorough analysis on how the parameter MAX_M influences the overall test results.

D. Enhancing ML-Driven: An Adaptive Approach to Balance Exploration and Exploitation

Maintaining a good balance between *exploration* and *exploitation* is extremely important for a successful application of EAs [49]. In our case, the balance is determined by the parameter MAX_M, which affects the number mutants (offsprings) generated from each selected test. For `ML-Driven D`, MAX_M is set to a higher value, which leads selecting fewer tests for mutation ($\approx\lambda/100$), but generating more mutants per selected test (higher exploitation). For `ML-Driven B`, MAX_M is set to a lower value, which leads selecting more tests for mutation ($\approx\lambda/10$), but generating fewer mutants per selected test (higher exploration). Notice that for both variants, the total test budget is the same, but the allocation of the test budget differs.

A detailed analysis presented in Section V-A shows that no variant is superior over the other [8]: `ML-Driven D` performs better at the beginning of the search but `ML-Driven B` outperforms it in later stages. The reason for this phenomenon is because of the number of bypassing tests selected and their influence on the overall performance. At the beginning, there are only a few available tests with a high bypassing probability. Due to the higher value of MAX_M, `ML-Driven D` is likely to select only these tests for mutation and, thus, exploits better the neighborhood of highly probable bypassing tests. This leads to generating more bypassing attacks in the earlier stages of the search. Reversely, since MAX_M is lower in `ML-Driven B`, more tests are selected for mutation, resulting in selecting not only the few tests with a high bypassing probability, but also tests with a low bypassing probability. As a result, `ML-Driven B` generates fewer new bypassing tests.

On the other hand, after some iterations, there are more tests with a high bypassing probability being available for mutation. In such a scenario, selecting more tests and mutating each one less often helps preserving diversity: `ML-Driven B` will explore the neighborhoods of multiple highly probable bypassing tests. `ML-Driven D` explores the neighborhoods of fewer tests while many other tests with similar bypassing probability remain unexplored.

In this section, we propose an improved variant of our attack generation strategy called `ML-Driven E` (Enhanced). Its goal is to combine the strengths of `ML-Driven B` and `ML-Driven D` by adaptively adjusting the parameter MAX_M to better balance *exploration* and *exploitation*. We propose `ML-Driven E`, which is a more flexible approach for assigning the test budget to individual tests. Instead of generating a fixed number of mutants per test, as done with `ML-Driven` B/D, the number of mutants is calculated dynamically. The rationale of `ML-Driven E` is twofold. First, the available test budget should be allocated only to tests with a high bypassing probability. Second, the test budget should be divided amongst all tests in proportion of that probability, thus favoring those more likely to yield new attacks.

Given the total number of offsprings λ to generate, a set T of tests selected as parents, the probability $P(t)$ of test t to bypass the WAF, then the number of offsprings m_t to generate for test t is defined as

$$m_t = \frac{P(t)}{\sum_{x \in T} P(x)} * \lambda. \tag{1}$$

On the right-hand side, the fraction represents the relative bypassing probability of t by dividing its bypassing probability with the sum of all bypassing probabilities of tests $x \in T$. By multiplying the relative bypassing probability of test t with λ (total number of offsprings), t is assigned a share of the total budget proportional to its relative bypassing probability. In other words, the number of offsprings/mutants to generate for each test t is proportional to its bypassing probability in relation to the probabilities of all selected parents.

`ML-Driven E` shares the same pseudocode with `ML-Driven D` and `ML-Driven B` with the exception of the routine OFFSPRINGSGEN used to generate offsprings in Algorithm 2. Instead of using Algorithm 3, `ML-Driven E` uses the new adaptive routine detailed in Algorithm 4, which includes the proposed modification to the budget calculation.

Similarly to Algorithm 3, Algorithm 4 generates λ offsprings within the loop in lines 3–18. The differences concern 1) the number test cases selected as parents and 2) the number of mutants/offsprings generated from each individual parent. In line 4, the routine $selectAttacksForMutation$ selects a set of attacks T that have the highest bypassing probability from all available attacks in the current population. All attacks above a

Algorithm 4: Adaptive Offsprings Generation.

```
1:  procedure AdaptiveOffspringsGen($population$, λ, $DT$)
2:    offsprings ← ∅
3:    while |offsprings| < λ do
4:      $T \leftarrow selectAttacksForMutation(parents)$
5:      for all $t \in T$ do
6:        $m_t \leftarrow getMutationBudget(t, T, DT)$
7:        $V \leftarrow getSliceVector(t)$
8:        $pathCondition \leftarrow getPath(V, DT)$
9:        while $m_t > 0$ do
10:         $s \leftarrow pickASliceFrom(V)$
11:         if $satisfy(s, pathCondition)$ then
12:           $newTest \leftarrow mutate(t, s)$
13:           $offsprings \leftarrow offsprings \cup newTest$
14:           $m_t \leftarrow m_t - 1$
15:         end if
16:       end while
17:     end for
18:   end while
19:   return offsprings
20: end procedure
```

configurable threshold σ, e.g., in our experiment $\sigma = 80\%$, are selected. The loop from line 5 to 17 is executed for each selected attack in T. Within the loop, in line 6, the routine $getMutationBudget$ calculates the number of offsprings/mutants m_t to generate from attack t with respect to T and DT, the latter being the classifier. In other words, this method implements (1). Lines 9–16 describe the mutation procedure for t, that is mostly unchanged from the original version of the algorithm, except that the number of generated offsprings/mutants for t is set to m_t.

Since the number of tests with a bypassing probability larger than 80% may vary across generations, both the number of selected parents and the budget allocation vary over time. When there are only few tests with a large bypassing probability ($\geq\sigma$), Algorithm 4 selects only those tests as parents. As a result, the number of offsprings m_t generated from each parent t will be large (higher exploitation). Instead, when the current population contains many tests with a large bypassing probability ($\geq\sigma$), the total budget of μ offsprings is shared among a larger set of parents. As a consequence, fewer mutants will be generated from each parent (higher exploration). Therefore, Algorithm 4 adaptively balances exploration and exploitation depending on the number tests in the current population that have a bypassing probability greater than σ.

IV. Empirical Study

This section evaluates the proposed testing strategies in two separate case studies: a popular open-source WAF and a proprietary WAF that protects a financial institution. Section IV-A introduces the case studies. Section IV-B formulates the research questions. Section IV-C explains the procedure we followed to execute the experiments, while Section IV-E describes the experiment variables.

A. Subject Applications

1) Open-Source WAF: In this case study, the firewall under test is *ModSecurity*, which implements the OWASP core rule set. ModSecurity is an open-source WAF that can be deployed with the Apache HTTP Server 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 OWASP core rules target various kinds of attacks, e.g., Trojan, Denial of Service, and SQLi, and are maintained by an active community of security experts.

The web applications under protection are HotelRS, Cyclos, and SugarCRM. HotelRS is a service-oriented based system, providing web services for room reservation. It was developed and used in [16]. Cyclos is a popular open-source Java/Servlet Web Application for e-commerce and online payment.[5] SugarCRM is a popular customer relationship management system.[6] 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 protects the application's web services from SQLi attacks. Specifically, since these web services receive SOAP messages[7] from web clients, a malicious client can seed an 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.

2) Proprietary WAF: In the industrial setting, we evaluate our approach on a proprietary WAF that is used in a corporate IT environment to protect back-end web services. These services are the backbone of a financial corporation and process thousands of transactions daily. To provide protection from malicious requests, the WAF validates incoming requests in two steps. First, the values in a request are validated with respect to data types (e.g., string or numeric) and boundary constraints, e.g., a credit card number is expected to be a sequence of 16 to 19 digits. In a second step, each value is checked to make sure that it does not contain known malicious string patterns (i.e., using an SQLi blacklist) commonly used in attacks. Only if the request passes both validation steps, the request is forwarded to the back-end services.

The computation time required to evaluate all testing strategies with the proprietary WAF is highly expensive because of the following:

1) the slow responsive time of the web services;
2) the number of testing strategies to compare;
3) the number of repetitions to perform for each testing strategy.

Therefore, for the experiment, we used of a high-performance cluster [2] when testing the proprietary WAF. Furthermore, we

[5] http://project.cyclos.org

[6] http://sourceforge.net

[7] http://www.w3.org/TR/soap12-part1

TABLE II
AVERAGE RESPONSE TIME IN MILLISECONDS OF SOME WEB SERVICE OPERATIONS IN OUR EXPERIMENTAL ENVIRONMENT COMPARED TO THE CASE STUDY'S ENVIRONMENT

	Op. 1	Op. 2	Op. 3	Op. 4
RTT_{HPC}	11, 36	11, 39	11, 46	16, 61
RTT_{ACT}	456, 56	180, 02	302, 09	854, 63

had to optimize a replica of the test environment to significantly decrease response times when invoking services.

In our optimized test environment, all configurations related to request filtering, that is whether a request is blocked or let through, are copied. Other configurations, e.g., logging or encryption, are disabled. The web application under protection is replaced by a simple mock-up application, which implements the same interface as the original application under protection. The mock-up replays a set of recorded responses from the original application and, thus, the WAF remains unaffected. Table II shows the message round trip time (RTT) per operation computed over a time span of 30 days with the actual environment (RTT_{ACT}) compared to the optimized environment on the HPC (RTT_{HPC}). As we can see, the time has been reduced significantly. Note that, even with these optimizations, the total computation time of our experiments is equivalent to 8 years, 337 days, and 12 h on a single CPU core.

Even though an optimized test environment is required from an experimental standpoint, in practice, testing the firewall is just a single test activity in an array of test activities (e.g., testing the WAF, services, front-end). Given time and resource constraints, creating and maintaining test environments that are specific for each single test activity is very costly. Based on our experience, we found that test engineers prefer test copies of the actual WAF configuration and services.

Since we, by design, optimized our test environment to enable large-scale experiments, the test execution times in our experimental setting is not representative of test execution times in the actual environment (see Table II). This is a problem as it biases the results of our experiments to the advantage of approaches that are less expensive in terms of test case generation but lead to the execution of more test cases, such as random testing. Therefore, in our analyses, we transform the time scale of the optimized environment into a realistic one, accounting for the actual test execution times in practice.

Assume $T = \{t_1, \ldots, t_n\}$ is a set of timestamps measured on the experimental environment, such that one timestamp is noted after the execution of every test. Then, for the ith test case execution, $f(t_i) = t_i + (\text{RTT}_{\text{ACT}} - \text{RTT}_{\text{HPC}}) * i$ transforms a timestamp $t_i \in T$ into the corresponding timestamp in the actual environment $f(t_i)$. We will use the latter time scale to compare test strategies in a realistic fashion.

B. Research Questions

This paper investigates several variants of a machine-learning-driven testing strategy and a random testing strategy. We compare and evaluate all these strategies for their capability of finding bypassing attacks on both subject applications, i.e., ModSecurity and the proprietary WAF.

Since all testing strategies generate attacks from the same input space, i.e., the grammar introduced in Section III-A, we evaluate how efficient the different strategies are in sampling the input space for bypassing attacks. Therefore, we measure for each strategy how many distinct bypassing attacks are found over time.

RQ1: *How efficient are ML-Driven E, ML-Driven B, ML-Driven D, and RAN in finding bypassing tests?*

To assess the impact of the machine learning algorithms on the test result, we implemented two alternative classifiers for each of the `ML-Driven` strategies: *RandomTree* and *RandomForest*. As detailed in Section III-C3, both algorithms have complementary advantages and drawbacks. In brief, RandomForest is an ensemble classifier and, hence, is expected to be more robust against changes in the training set than *RandomTree*. However, *RandomForest* comes at a higher computational cost than *RandomTree*, since multiple models have to be learned. To assess the influence of both algorithms on our approach, we run `ML-Driven` with both algorithms and compare the number of identified bypassing attacks and path conditions in a given time budget.

RQ2: *Does the choice of machine learning algorithm matter?*

We evaluate whether, in our context, the benefits of the RandomForest compared to the RandomTree justify the increased computation overhead to learn the classifier. Therefore, we compare how many bypassing tests are found over time with these two algorithms. In addition, we assess whether the algorithms have an impact on the stability of the test result, i.e., we compare the variation among repetitions of the same test run.

RQ3: *How does ML-Driven compare to similar techniques?*

RQ3 compares our proposed technique with existing techniques for testing WAFs. We perform a quantitative comparison of the techniques by comparing the number of bypassing attacks found by each technique.

RQ4: *Are we learning new, useful attack patterns as the number of distinct, bypassing attacks increases?*

RQ4 assesses whether, as we find more bypassing tests, we also identify more attack patterns that can be useful to improve the rule set of the WAF. In our context, an attack pattern is the underlying root cause that enables an attack to bypass the WAF. For example, the attacks `union *!50000*select pwd from user` and `union *!50000*select 99` share the pattern `union *!50000*select` (root cause) while the remainder of the attacks differ. We want to investigate whether identifying successful attack patterns help understanding why attacks are bypassing and eventually fix the WAF to correctly detect further attacks. For the `ML-Driven` techniques, such a pattern is characterized by a path condition (see

Definition 3). A path condition characterizes the slices, or combination of slices, that are likely causing an attack to bypass. Thus, a path condition represents a pattern that is not correctly detected by the WAF.

To answer RQ4, we measure how many path conditions can be extracted from a model that is learned by the `ML-Driven` techniques. More specifically, we analyze the growth in the number of path conditions as the number of successful, distinct attacks grows over time.

C. Procedure

We implemented the techniques proposed in this paper into our SQLi testing tool called *Xavier*. *Xavier* supports the automated testing of web services for SQLi vulnerabilities and has been described in [10]. `ML-Driven` generates test cases in the form of SQLi attack strings, such as `'␣OR"1"="1"#`. To generate malicious requests, *Xavier* takes such attack strings and injects them into sample SOAP messages, which are subsequently sent to an application under test. *Xavier*, therefore, relies on sample SOAP messages as inputs. They can be taken from existing web service test suites, or can easily be generated from the WSDL[8] that describes the service interface under test.

In our experiments, when available, we use SOAP messages from the functional test suite of the service under test (Cyclos and our industrial case study) or, otherwise, we manually create SOAP messages from the WSDLs (HotelRS, SugarCRM). Each of these messages consists of a number of parameters and their legitimate values, which the services expect and that the WAF has to let through. In our testing process, each SOAP message is considered separately. A test generation technique, `ML-Driven` or RAN, continuously generates attacks, injecting one attack each time into a parameter of the selected SOAP message to create a new SOAP message, and then sends it 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, and otherwise are blocked. In case a request is blocked, the WAF replies to the client that 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" (for blocked), and otherwise a passed label "P" (bypassing).

D. Parameter Setting

For the `ML-Driven` approaches, there are various parameters to set for both the machine learning algorithms and (μ + λ)-EA. For most of the parameters (e.g., machine learning training setting), we follow the recommendations from the literature [15], [28], [41]. Moreover, we used a trial-and-error procedure to calibrate the parameters μ and λ. The final parameter values are as follows:

1) Population Size (μ): Typically, the population size for EAs can vary from tens to several thousands of individuals [41]. The best setting for this parameter depends on the nature of the problem and on the size of the search space [31]. In our case, the population size determines the number of tests that will be preserved for the next generation. A small population size may lead to loss of test cases with large bypassing probability, thus reducing the chances of generating new attacks able to pass through the WAF. From our preliminary experiments, we observed that a population size of 500 individuals works best for `ML-Driven` since it helps preserving as many as possible probable bypassing attacks.

2) Number of Offsprings (λ): In `ML-Driven`, newly generated offsprings (test cases) are used to update the *archive*, which is later used to retrain the classifier DT by applying either *RandomTree* or *RandomForest*. Therefore, offsprings are the additional "data points" that are used for updating DT. A large λ value means that in each generation a significant portion of the training set consists of new test cases; a small λ value leads to retraining DT with almost the same training set (i.e., updating DT would be less effective). In this paper, we set $\lambda = 4000$ to guarantee that DT is always retrained with a large portion new test cases. While the total number offsprings is fixed, the number of offsprings generated from each test t differs for the three variants of `ML-Driven`. For `ML-Driven B` and `ML-Driven D`, it is fixed and it is equal to $\text{MAX}_M = 10$ and $\text{MAX}_M = 100$, respectively. In `ML-Driven E`, the number of offsprings for each test is determined dynamically using the adaptive strategy described in Section III-D.

3) Archive Size: The archive keeps track of all attacks generated across the generations and it is used at the end of each generation as training set for machine learning. To avoid exceeding the resources of a typical laptop and to finish in a reasonable time, we limit the training set to 6000 blocked attacks and 6000 bypassing attacks. If the number of tests in the archive is larger, we consider only the most recent tests, i.e., those produced in latest generations.

4) Stopping Condition: The search terminates when the maximum search budget of 175 min is reached for ModSecure. For the proprietary WAF, we used a larger search budget of 500 min. To allow a fair comparison, RAN was configured with the same stopping condition.

5) Machine Learning Setting: For the machine learning algorithms used in this paper, i.e., *RandomTree* and *RandomForest*, we used their implementations available in `Weka` [28] with the default parameter values.

6) Number of Repetition: To account for the randomization involved in the testing techniques (e.g., RAN), each algorithm was run ten times for each subject application.

E. Variables

The following dependent variables are controlled or measured in our experiments.

1) $\boldsymbol{D_t}$: The number of distinct tests that can bypass a target WAF at time t is a way to measure the efficiency of a test strategy. Note that, given two distinct tests in the same attack category, one might be caught by the WAF while the other bypasses it. 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 rule set,

[8] [Online]. Available: http://www.w3.org/TR/wsdl

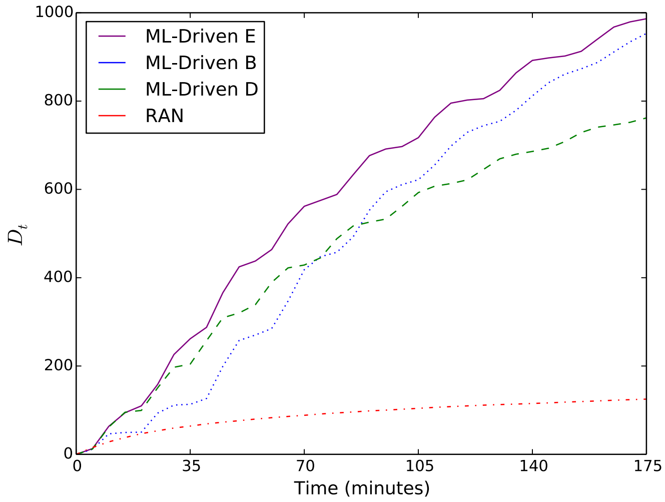

Fig. 5. Average number of bypassing tests (D_t) found for nine tested operations (ten repetitions each) for ModSecurity.

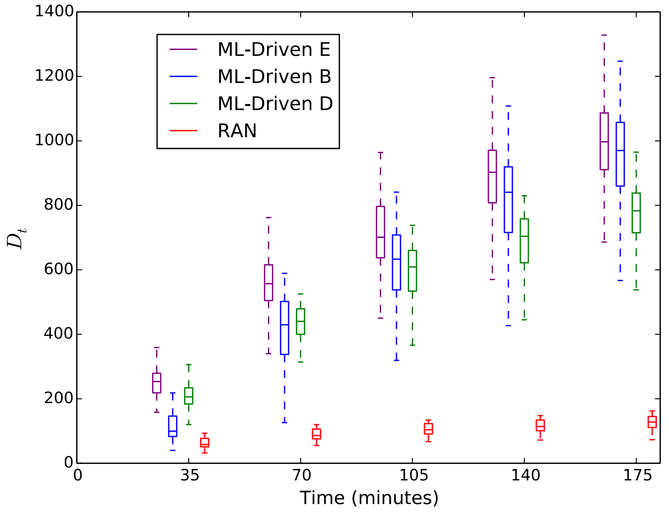

Fig. 6. Statistical variation for the number of bypassing tests found (D_t) across all tested parameters for ModSecurity.

or by an identical rule that is not general enough to block all tests in a category. Therefore, identifying similar but distinct bypassing tests is useful to identify attack patterns.

2) $\boldsymbol{D_{pc}}$*:* The number of distinct path conditions that can be extracted from DTs. Each path condition characterizes a string pattern that a group of bypassing attacks has in common. Such string patterns can be added to the WAF rules to prevent further attack attempts containing the same pattern. Hence, D_{pc} is a measure for how many attack patterns have been uncovered.

For each subject application, we ran each testing strategy ten times and collected the number of distinct bypassing tests ($\boldsymbol{D_t}$) and distinct path conditions ($\boldsymbol{D_{pc}}$) in each run. To this aim, every time a new test t was generated, we collected the passing wall-clock time since the beginning of the search and then executed t to see whether or not it could bypass the WAF. Therefore, we can compare the values of $\boldsymbol{D_t}$ and $\boldsymbol{D_{pc}}$ collected over time.

To verify whether $\boldsymbol{D_t}$ scores significantly differ when using two different testing strategies (e.g., RAN and `ML-Driven E`), we used the Wilcoxon test with a significance level of $\alpha = 0.05$. The Wilcoxon test is a nonparametric test and, thus, does not require the data to be normally distributed. For the statistical analysis, we collected the $\boldsymbol{D_t}$ scores achieved by each alternative testing approach after intervals of 35 min for each repetition, resulting in five time windows for each run.

V. Results

In this section, we describe the results obtained in our case studies regarding the research questions formulated in Section IV-B.

A. RQ1: How Efficient are ML-Driven E, ML-Driven B, ML-Driven D, and RAN in Finding Bypassing Tests?

To answer *RQ1*, we applied the testing techniques to both subject applications and measured how many bypassing tests were found over time (D_t). In particular, we compared the performance of the techniques based on the cumulative number of distinct bypassing tests generated over time. In the following, we discuss the results for ModSecurity and the proprietary WAF separately.

TABLE III
Results of the Wilcoxon Test for ModSecurity

Time Window	RAN ($p < 0.01$)	ML-B ($p < 0.01$)	ML-D ($p < 0.01$)	ML-E ($p < 0.01$)
35 min		RAND	RAND ML-B	RAND ML-B ML-D
70 min		RAND	RAND	RAND ML-B ML-D
105 min		RAND	RAND	RAND ML-B ML-D
140 min		RAND ML-D	RAND	RAND ML-B ML-D
175 min		RAND ML-D	RAND	RAND ML-B ML-D

For each testing strategy (e.g., ML-E), we report whether it statistically outperforms its counterparts (e.g., RAN) when p-values < 0.01.

1) Results for ModSecurity: For the open-source WAF, we randomly selected nine parameters in total for testing, three parameters from each web application (HotelRS, SugarCRM, and Cyclos). For each technique, Fig. 5 depicts the average number of distinct, bypassing tests generated over time for ModSecurity measured within intervals of 5 min. The testing results for each individual parameter are in a technical report [8]. Fig. 6 depicts the same data as boxplots to help visualize statistical variation across the ten repetitions. Table III reports the results of the Wilcoxon test by listing, in each cell, the test strategies that are significantly outperformed by the strategy matching the corresponding column.

The first observation is that all techniques can generate tests that bypass the WAF, suggesting that the WAF does not provide complete protection from SQLi attacks, putting online systems under its protection at risk. Furthermore, by observing executed SQL statements on the database, we found that these bypassing tests can exploit SQLi vulnerabilities in HotelRS and SugarCRM. Second, the sharply increasing plots corresponding to `ML-Driven E`, `ML-Driven B` , and `ML-Driven D` indicate that they are much more efficient than RAN, the baseline for comparison. Overall, the results show that the `ML-Driven` techniques outperform RAN by an order of magnitude with respect to the number of distinct bypassing tests generated. The Wilcoxon test revealed that these differences were always statistically significant as depicted in Table III (p-values < 0.01).

Among the machine-learning-driven techniques, `ML-Driven E` constantly finds the most bypassing tests compared to `ML-Driven B` and `ML-Driven D`, which suggests that the adaptive budget allocation works well. The differences between `ML-Driven E` and its predecessors are always statistically significant in all the five time windows. One issue with `ML-Driven` D/B is that at the beginning of the test run, `ML-Driven D` finds more bypassing tests, while this is the opposite later in the test run. More specifically, after a time window of 35 min, `ML-Driven D` is statistically superior to `ML-Driven B` while between 70 and 105 min the two `ML-Driven` approaches are statistically equivalent; finally, in the last two time windows (i.e., 140 and 175 min), `ML-Driven B` statistically outperforms `ML-Driven D`. Since both techniques implement the same algorithm, this phenomenon can be attributed to a difference in the choice of parameters, or, more precisely, the parameter that determines the number of mutants generated per test (a detailed analysis is provided in a technical report [8]). While `ML-Driven D` and `B` generate a fixed number of mutants per test, `ML-Driven E` adjusts the number of mutants in proportion to the test's bypassing probability. As a result, `ML-Driven E` spends the test budget more efficiently and finds bypassing attacks faster.

The plots for the `ML-Driven` techniques are also slightly oscillating, thus depicting the effect of iterative retraining of the classifier. The flat segments match the time intervals where the classifier is recomputed and no new tests are generated. The slopes of the plots tend to decrease over time as it becomes increasingly harder to find new bypassing tests that have not yet been executed.

2) Results for the Proprietary WAF: We now address *RQ1* for the second subject application, the proprietary WAF. As for ModSecurity, all techniques are evaluated measuring the number of bypassing tests that are found over time.

Due to the considerably higher complexity of the WAF in the industrial case study, we selected a higher number of parameters for testing. Given that all testing strategies have to be applied to each parameter and each test run was repeated ten times, testing all parameters would have been infeasible. Therefore, we selected one parameter for each distinct data type in the WSDL of the services under test, which resulted in a total of 75 parameters. The WAF in this case study determines the input validation routine to be executed for a parameter based mainly on the corresponding data type; hence, selecting one parameter per data type maximizes the coverage of input validation routines.

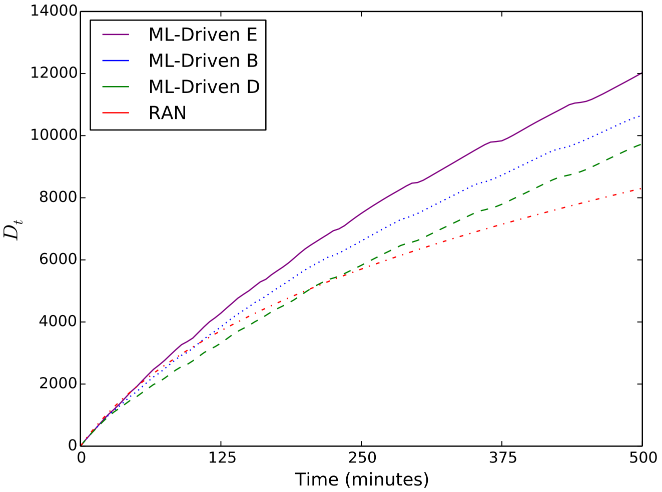

Fig. 7. Average number of bypassing attacks, proprietary WAF, all tested parameters.

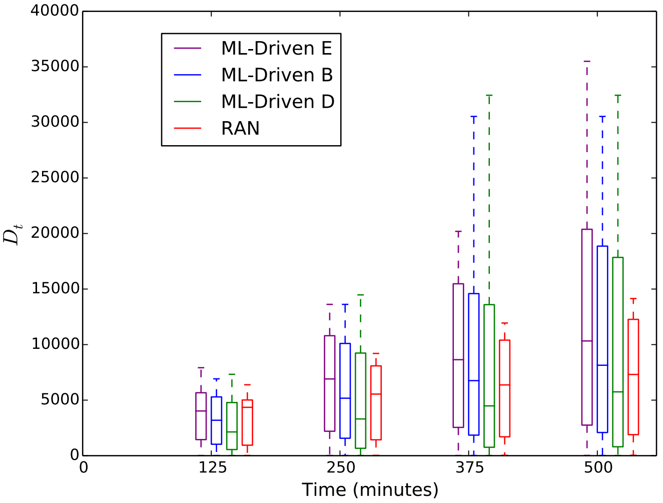

Fig. 8. Boxplots of the number of bypassing tests, all tested parameters, proprietary WAF.

TABLE IV
GROUPS OF PARAMETERS WITH A SIMILAR INPUT CONSTRAINT

	#Parameter	Input Constraint
Group 1	2	Up to 8 Char.
Group 2	7	Up to 16 Char.
Group 3	8	Up to 25 Char.
Group 4	12	Up to 35 Char.

Out of the 75 tested parameters, bypassing tests could be generated for 29 parameters. Each testing technique was able to generate bypassing tests for all of these 29 parameters. Fig. 7 depicts the average number of distinct bypassing tests per test strategy. The average is computed from all tested parameters and ten repetitions per parameter.

Out of all techniques, `ML-Driven E` generates the most bypassing tests on average, followed by `ML-Driven B` and `ML-Driven D`. RAN finds the least bypassing tests. Since the statistical variation in the plots is high (see boxplot in Fig. 8), we

TABLE V
RESULTS OF THE WILCOXON TEST FOR THE PROPRIETARY WAF

Group	Time Window	RAN	ML-B	ML-D	ML-E
1	35 min		ML-D ($p = 0.01$)		
	70 min	ML-D ($p = 0.01$)	ML-D ($p = 0.03$)	RAND ($p < 0.01$)	
	105 min	ML-B ($p = 0.03$) ML-D ($p = 0.01$) ML-E ($p < 0.01$)	ML-D ($p = 0.04$)		
	140 min	ML-B ($p < 0.01$) ML-D ($p < 0.01$) ML-E ($p < 0.01$)			
	175 min	ML-B ($p < 0.01$) ML-D ($p < 0.01$) ML-E ($p < 0.01$)			
2	35 min	ML-D ($p < 0.01$)	RAND ($p < 0.01$)		RAND ($p < 0.01$) ML-B ($p < 0.01$) ML-D ($p < 0.01$)
	70 min	ML-D ($p < 0.01$)	RAND ($p < 0.01$)		RAND ($p < 0.01$) ML-B ($p < 0.01$) ML-D ($p < 0.01$)
	105 min	ML-D ($p < 0.01$)	RAND ($p < 0.01$)		RAND ($p < 0.01$) ML-B ($p < 0.01$) ML-D ($p < 0.01$)
	140 min	ML-D ($p < 0.01$)	RAND ($p < 0.01$)		RAND ($p < 0.01$) ML-B ($p < 0.01$) ML-D ($p < 0.01$)
	175 min	ML-D ($p < 0.01$)	RAND ($p < 0.01$)		RAND ($p < 0.01$) ML-B ($p < 0.01$) ML-D ($p < 0.01$)
3	35 min	ML-B ($p < 0.01$) ML-D ($p < 0.01$)			RAND ($p < 0.01$) ML-B ($p < 0.01$) ML-D ($p < 0.01$)
	70 min	ML-B ($p < 0.01$) ML-D ($p < 0.01$)			RAND ($p < 0.01$) ML-B ($p < 0.01$) ML-D ($p < 0.01$)
	105 min	ML-B ($p < 0.01$) ML-D ($p < 0.01$)			RAND ($p < 0.01$) ML-B ($p < 0.01$) ML-D ($p < 0.01$)
	140 min	ML-B ($p < 0.01$) ML-D ($p < 0.01$)			RAND ($p < 0.01$) ML-B ($p < 0.01$) ML-D ($p < 0.01$)
	175 min	ML-B ($p < 0.01$) ML-D ($p < 0.01$)			RAND ($p < 0.01$) ML-B ($p < 0.01$) ML-D ($p < 0.01$)
4	35 min		RAND ($p < 0.01$) ML-D ($p < 0.01$)	RAND ($p < 0.01$)	RAND ($p < 0.01$) ML-B ($p < 0.01$) ML-D ($p < 0.01$)
	70 min		RAND ($p < 0.01$) ML-D ($p < 0.01$)	RAND ($p < 0.01$)	RAND ($p < 0.01$) ML-B ($p < 0.01$) ML-D ($p < 0.01$)
	105 min		RAND ($p < 0.01$) ML-D ($p < 0.01$)	RAND ($p < 0.01$)	RAND ($p < 0.01$) ML-B ($p < 0.01$) ML-D ($p < 0.01$)
	140 min		RAND ($p < 0.01$) ML-D ($p < 0.01$)	RAND ($p < 0.01$)	RAND ($p < 0.01$) ML-B ($p < 0.01$) ML-D ($p < 0.01$)
	175 min		RAND ($p < 0.01$) ML-D ($p < 0.01$)	RAND ($p < 0.01$)	RAND ($p < 0.01$) ML-B ($p < 0.01$) ML-D ($p < 0.01$)

For each testing strategy (e.g., ML-E), we report whether it statistically outperforms its counterparts (e.g., RAN) as well the corresponding p-values.

separate the parameters into groups to better analyze the results. Parameters in the same group share similar input constraints in terms of number of allowed characters and tend to produce a similar number of bypassing test cases. It is worth noticing that this particular WAF considers not only such input constraints but also other criteria (e.g., SQLi blacklist). Therefore, parameters in a same group do not necessarily produce the same results. Table IV shows the groups: Group 1 has two parameters that share an input constraint that restricts the number of characters to a maximum of eight. Similarly, Groups 2, 3, and 4 have similar constraints with 16, 25, and 35 characters, respectively. For each group, Table V reports the results of the Wilcoxon test when comparing the number of bypassing attacks generated by the different testing strategies.

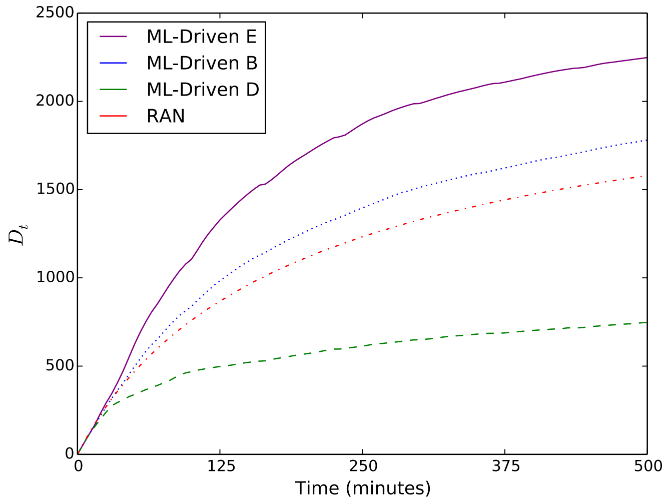

Fig. 9. Group 2: Average number of bypassing attacks, proprietary WAF, seven parameters.

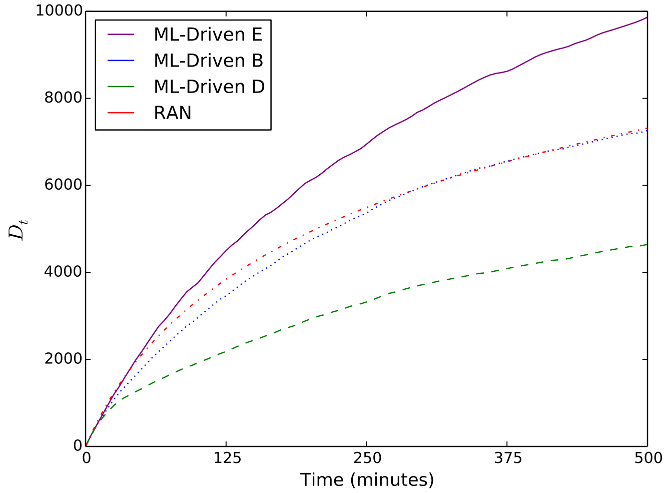

Fig. 10. Group 3: Average number of bypassing attacks, proprietary WAF, eight parameters.

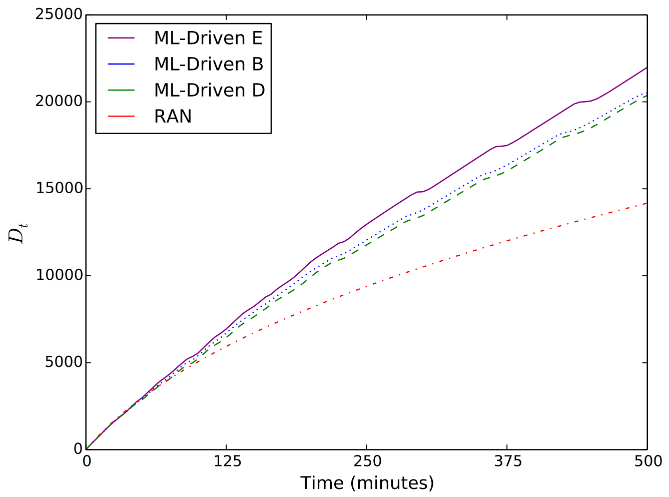

Fig. 11. Group 4: Average number of bypassing attacks, proprietary WAF, 12 parameters.

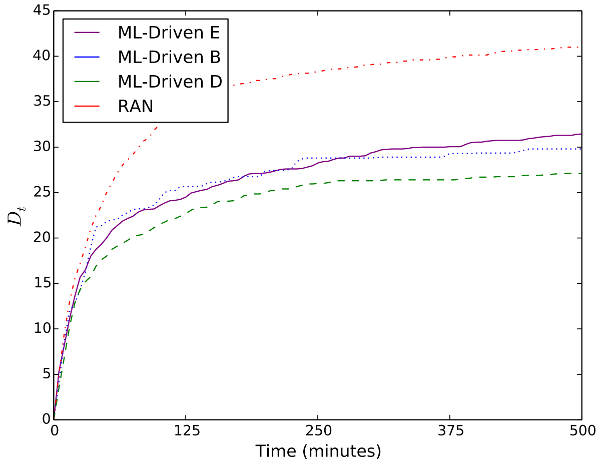

Fig. 12. Group 1: Average number of bypassing attacks, proprietary WAF, two parameters.

According to Table V, we observe that for Groups 2, 3, and 4 `ML-Driven E` is always significantly superior to its predecessor as well as to RAN. However, looking at Figs. 9–11, we notice the following trend: the more the bypassing tests, the smaller the difference in test results across `ML-Driven` techniques. For example, `ML-Driven E` generates, on average, +25% bypassing attacks compared to the second best approach in the comparison (i.e., `ML-Driven B`) for Group 2. For Group 3, the gap between `ML-Driven E` and the closest alternative approach (i.e., RAN) is +30% on average. For Group 4, the average difference between the `ML-Driven E` and its predecessor is lower than 10%.

The average number of bypassing tests for the two parameters in Group 1 is depicted in Fig. 12. All techniques tend to saturate after about 125 min and, after that, the number of bypassing tests increases very slightly. RAN is the most effective approach, significantly outperforming all `ML-Driven` variants after 105 min of running time (see Table V). At the end of the search, RAN reaches up to approximately 40 bypassing tests while all the `ML-Driven` techniques find around 30 bypassing tests. The most prominent difference of this group, as compared to the others, is that very few bypassing tests are found. This is due to the fact that the number of allowed characters is only eight, thus resulting in a small number of attacks to bypass. Examining the training sets used to learn the classifiers more closely, we find that the training sets are heavily imbalanced. There are only between 15 and 30 bypassing attacks compared to up to 6000 blocked attacks, i.e., they represent only 0.25%–0.5% of the data points in the training set. For the other tested parameters, where the machine-learning-driven techniques outperform RAN, there are typically several hundreds of bypassing attacks in a training set. As a result of the imbalanced training data, the learned classifier has a low recall (≈80%) and is less accurate in predicting an attack's bypassing probability. *RandomTree* and *RandomForest* tend to generate a constant classifier in such a scenario, i.e., the resulting classifier likely labels all data tests as "blocked" because this leads to a very low classification error ($<1\%$).

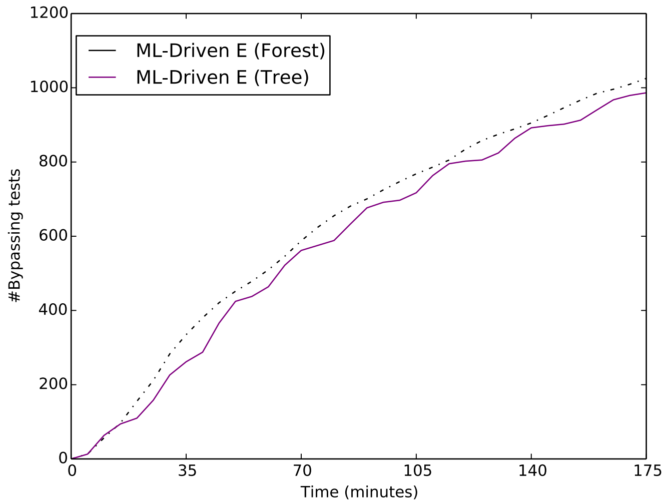

Fig. 13. Comparison of the RandomTree and RandomForest classifier. Average number of bypassing tests found for nine tested operations (ten repetitions each) in ModSecurity.

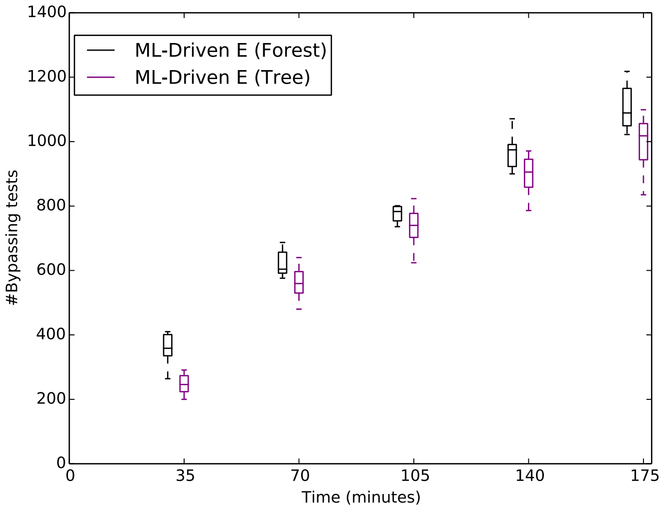

Fig. 14. Boxplots for the number of bypassing tests found with one representative parameter (get-relationships) in ModSecurity.

Moreover, if we analyze the training sets used in consecutive retraining iterations (generations), we find that there are not substantially more bypassing attacks added to the training set. The quality of the classifier does not improve over the subsequent generations and is hardly worth the computation time necessary for retraining. To address this shortcoming, we conclude that the classifier should only be retrained if the training data have sufficiently improved. Otherwise, the existing classifier may be used to generate more offsprings until more bypassing attacks are found or the test budget is exceeded. In addition, in the presence of high imbalanced training sets, we may apply various well-known strategies in machine learning, such as data resampling strategies, weighted training, and penalty-based training. Investigating the best strategy to address the imbalance problem for `ML-Driven` is a part of our future agenda.

B. RQ2: Does the Choice of Machine Learning Algorithm Matter?

To answer *RQ2*, we run `ML-Driven E` with the chosen machine learning algorithms, namely, RandomTree and RandomForest, and we compare the results. For this analysis, we focus on `ML-Driven E` only since it is the most efficient variant of `ML-Driven` according to the results discussed in Section V-A. The evaluation is performed on the open-source case study (ModSecurity) in the same fashion as for RQ1.

Fig. 13 shows the average number of bypassing tests for nine selected parameters and ten repetitions. The result indicates that `ML-Driven E` with RandomForest finds slightly more bypassing tests than `ML-Driven E` with RandomTree. However, the difference is not practically significant. The advantage of using a more stable classifier is partially lost due to the increased computation time for constructing this classifier. Therefore, although RandomForest may generate a more accurate/stable classifier than RandomTree, its overhead negatively affects the number of generations (i.e., the number of tests to run against the WAF) that can be performed within the same amount of time (search budget). In other words, the gain in accuracy when using RandomForest does not compensate its overhead when used inside the test case generation loop. Even if less precise, RandomTree enables more iterations and an increased number of new test executions.

Fig. 14 shows a sample boxplot corresponding to the observed statistical variation for one representative parameter over ten repetitions. We can see that the variation is small and similar for RandomForest and RandomTree. Similar results are observed for the other parameters. Therefore, to conclude, the choice between the two considered machine learning algorithms has no significant impact with respect to the number of bypassing tests or the degree of variation among repetitions.

C. RQ3: How Does ML-Driven Compare to Similar Techniques?

To answer *RQ3*, we compare our proposed technique `ML-Driven` with similar tools. The tools considered in this comparison are chosen according to the following selection criteria: similar objective as `ML-Driven`, i.e., finding SQLi attacks that bypass a WAF under test; state-of-the-art covering a range of SQLi attacks and obfuscation methods; widely used or developed by known institutions or security experts. Following these criteria, we selected the *WAF Testing Framework* and *SqlMap* for comparison.

The WAF Testing Framework[9] tests the attack detection capabilities of a WAF under test. It is a state-of-the-art tool developed by Imperva, which is the developer of a proprietary WAF and several other security solutions. The tool comes with a number of malicious HTTP requests that contain common SQLi attacks. To test a WAF, it submits the HTTP requests one by one to the WAF and checks if the malicious requests are blocked correctly.

SqlMap[10] is a well-known penetration testing tool to detect SQLi vulnerabilities in web applications and services. It generates a large and diverse set of SQLi attacks, which cover all common SQLi attack techniques. To evade detection of WAFs,

[9] Downloaded from https://www.imperva.com/Resources/FreeEvaluationTools

[10] Version 1.0.7.42 downloaded from https://sqlmap.org

TABLE VI
TAMPER SCRIPTS OF SQLMAP SELECTED FOR OUR EXPERIMENTS

Tamper Script Name	Short Description
between.py	Inserts a clause with the SQL keyword BETWEEN
commalesslimit.py	Replaces instances like “LIMIT M, N” with “LIMIT N OFFSET M”
commalessmid.py	Replaces instances like “MID(A, B, C)” with “MID(A FROM B FOR C)”
concat2concatws.py	Replaces instances like “CONCAT(A, B)” with “CONCAT_WS(MID(CHAR(0), 0, 0), A, B)”
equaltolike.py	Replaces all occurrences of operator equal (“=”) with operator “LIKE”
greatest.py	Replaces greater than operator (“>”) with “GREATEST” counterpart
halfversionedmorekeywords.py	Adds versioned MySQL comment before each keyword
ifnull2ifisnull.py	Replaces instances like “IFNULL(A, B)” with “IF(ISNULL(A), B, A)”
informationschemacomment.py	Adds a comment to the end of all occurrences of “information_schema” identifier
lowercase.py	Changes each keyword character to lower case
modsecurityversioned.py	Surrounds the attack string with a MySQL version-specific comment (randomly generated version number)
modsecurityzeroversioned.py	Surrounds the attack string with a MySQL version-specific comment (version number set to 0)
multiplespaces.py	Adds multiple spaces around SQL keywords
randomcase.py	Changes randomly the case of letters in an SQL keyword
randomcomments.py	Add random comments to SQL keywords
unionalltounion.py	Replaces UNION ALL SELECT with UNION SELECT
uppercase.py	Changes each keyword character to upper case
versionedmorekeywords.py	Surrounds each keyword with a version-specific MySQL comment

SqlMap uses several *tamper scripts*. Each tamper script applies a specific obfuscation method to an SQLi attack (e.g., randomly changes the case of letters in an SQL keyword). Since some tamper scripts are only meant to be applied with particular WAFs, web application frameworks (e.g., ASP or PHP), or database management systems, we selected a number of tamper scripts that are applicable to our subject applications (i.e., applicable to the proprietary WAF/ModSecurity, SOAP web services written in PHP/Java, and MySQL). Table VI lists the tamper scripts selected for our experiments and gives a brief description for each. Note that SqlMap does not report how many or which attacks bypassed a WAF under test. Therefore, we routed the HTTP traffic between SqlMap and the WAF under test through a custom HTTP proxy, which determines whether each generated attack bypassed the WAF or was blocked.

To compare `ML-Driven` with the WAF Testing Framework and SqlMap, we use both tools to test ModSecurity and our industry partner's proprietary WAF. The configuration of both WAFs is the same as in the other reported experiments. Since SqlMap's attack generation is randomized, we repeated each experiment involving SqlMap ten times and report the averages. The attack generation in the WAF Testing Framework is deterministic and, hence, we report the result of a single run. We do not impose any limit on the number of generated test cases; both tools are configured to execute as many test cases as they are capable of generating.

For the experiment with the proprietary WAF, we randomly selected one representative parameter to test from each of the four parameter groups (refer to Table IV). We found in the evaluation of RQ1 that parameters in a same group share a similar number of bypassing attacks. Given how computation intensive these experiments are, we test only a representative parameter of each group and interpret the result as a performance indicator for the other parameters in the same group. Table VII reports for both tools the total number of generated test cases ($\mathrm{TC}_{\mathtt{Total}}$) and the number of test cases bypassing the proprietary WAF ($\mathrm{TC}_{\mathtt{Bypass}}$). The WAF Testing Framework executes 19 test cases for each tested parameter. For the tested parameter of *Group 4*, three of the 19 attacks are bypassing. For all other tested parameters, the WAF Testing Framework does not find a bypassing attack. SqlMap generates a similar number of test cases for each tested parameter (on average between 3260.5 and 3283.9 test cases). For the parameter of *Group 4*, SqlMap finds, on average, 66.5 bypassing attacks and for the parameter of *Group 3*, SqlMap finds, on average, only 0.3 bypassing attacks. For the other parameters, SqlMap does not find any bypassing attacks.

TABLE VII
RESULTS FOR TESTING THE PROPRIETARY WAF WITH THE WAF TESTING FRAMEWORK AND SQLMAP

Parameter	WAF Testing Framework		SqlMap	
	#$\mathrm{TC}_{\mathtt{Total}}$	#$\mathrm{TC}_{\mathtt{Bypass}}$	#$\mathrm{TC}_{\mathtt{Total}}$	#$\mathrm{TC}_{\mathtt{Bypass}}$
Group 1	19	0	3283.9	0.0
Group 2	19	0	3281.6	0.0
Group 3	19	0	3278.9	0.3
Group 4	19	3	3260.5	66.5

The fact that both tools find the most bypassing attacks for Group 4 can be explained with the fact that the input length constraint for this group is the least strict (refer to Table IV). This also confirms the trend observed in the results for `ML-Driven`, which found the most bypassing attacks for Group 4 as well. For the other groups, the WAF Testing Framework and SqlMap do not reliably find bypassing attacks, because the generated attacks are either too long to satisfy the input length constraint or contain a pattern that is blacklisted.

For the experiment with ModSecurity, we use the WAF Testing Framework and SqlMap to test the same nine parameters as selected for the evaluation of `ML-Driven` (refer to RQ1). The WAF Testing Framework executes again 19 test cases per tested parameter and SqlMap executes a similar number of test cases as in the previous experiment (~3270 test cases per parameter). However, neither the WAF Testing Framework nor SqlMap find any bypassing attack for any of the tested parameters.

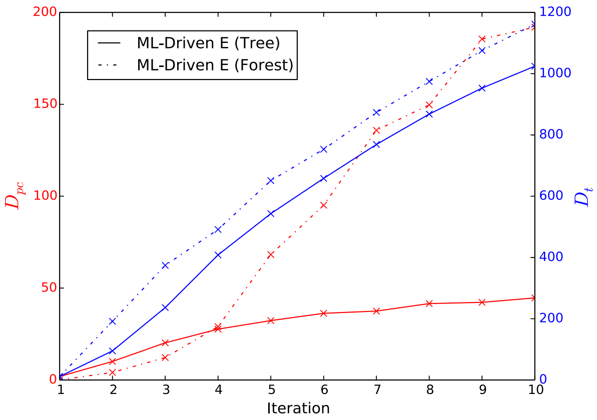

Fig. 15. Number of path conditions (red y-axis on the left) and bypassing tests (blue y-axis on the right) for `ML-Driven E` with RandomTree and RandomForest.

When comparing the results of the WAF Testing Framework and SqlMap with `ML-Driven` (refer to Section V-A), we find that the formers are either completely ineffective or significantly less effective than `ML-Driven`. For most of the tested parameters, both tools fail to find any bypassing attack, while `ML-Driven` finds a considerable number of them. For the few parameters, for which the alternative tools find bypassing attacks, `ML-Driven` finds an order of magnitude more bypassing attacks. Furthermore, the attacks found by the alternative tools are a subset of the attacks found by `ML-Driven`, and hence, using the alternative tools in combination with `ML-Driven` would not provide more distinct bypassing attacks. Another advantage of `ML-Driven` over the alternative tools is that `ML-Driven` abstracts path conditions from the bypassing attacks, which helps the firewall administrator to repair the WAF (refer to Section VI).

D. RQ4: Are We Learning New, Useful Attack Patterns as the Number of Distinct, Bypassing Attacks Increases?

As explained in Section III-C, the `ML-Driven` techniques periodically build a model (i.e., DTs) that is used to guide the generation of new attacks. However, this model can also be used to abstract common string patterns shared by and possibly causing bypassing attacks. Therefore, for RQ4, we analyze the models that are learned during the test runs of `ML-Driven E`. For both RandomForest and RandomTree, we analyze how the number of path conditions increases as the result of generating more distinct bypassing attacks.

Fig. 15 depicts the average number of path conditions that can be extracted from a model (red) and the number of bypassing tests with which the model is trained (blue). Since the `ML-Driven` techniques retrain the model at the end of each generation, the corresponding training iterations (or EA generations) are depicted on the horizontal axis. These curves are based on the average of all test runs performed with the ModSecurity case study.

The number of distinct, bypassing tests, which are used to train the model, is steadily increasing for both techniques. This is due to fact that the further a test run progresses, the more bypassing tests are found and are used, in turn, to retrain the model. The number of path conditions that can be extracted from a model is also steadily increasing for both techniques, although beginning from generation 4, there are significantly more path conditions extracted with RandomForest than with RandomTree. For RandomForest, the number of path conditions is close to 200 after 10 generations, compared with 50 for RandomTree.

TABLE VIII
SLICE ENCODINGS

Slice	Literal	Slice	Literal
$S1d$	or	Sb	!
$Sf1$	or␣1	$S3$	1
$S5$	/**/	$S6$	~␣1
$S26$	#	$S57$	␣1
$S2$	␣	$S29$	␣0
$S0$	0	$S15$	)
Sf	true	Sc	”

TABLE IX
DEMONSTRATION OF THE PATTERN LEARNED FROM EACH PATH CONDITION AND EXAMPLE ATTACKS CONTAINING THE PATTERN (THE PATTERN CHARACTERIZED BY THE PATH CONDITIONS IS HIGHLIGHTED IN RED)

Generations	Id	Pattern	Example SQLi Attack
1	pc_1	*or ␣ 1, /**/*	0 or␣1/**/
5	pc_2	*or, ␣1*	‘or true like␣1 ‖’
5	pc_3	*or,), #*	‘) or not false#
5	pc_4	*or, ␣0*	1 or true > (␣0)
5	pc_5	*or, true, #*	‘or~true#
5	pc_6	*or, 0*	0) or ! ”−−
5	pc_7	*or, ”*	“ ‖ true or”
5	pc_8	*or, /**/, #, ␣, 0, !*	‘␣or/**/!0 is true#
5	pc_9	*or, /**/, 1, ~␣1*	1 or/**/ 0 < (~␣1)

For both techniques, the number of bypassing tests used to train the model and the number of path conditions obtained from the model are concurrently increasing. This is no surprise as more test cases lead to refined models. For RandomForest, that leads to even more path conditions since many more trees are built.

In general, path conditions can help determine the reason why a set of attacks is bypassing. This knowledge can be utilized to stop further attack attempts from succeeding by inferring a string pattern from the path condition that matches all the attacks described by it. Such a string pattern can be in turn added to the WAF's rule set to block all further attacks containing the same string pattern.

To better explain the benefits of having more attack patterns, let us describe examples of path conditions and the attacks they characterize that are obtained from a test run of the industrial WAF used in our evaluation. In particular, let us consider an example of a path condition: $pc_1 = S1d \wedge Sf1 \wedge S5 \wedge \neg S11 \wedge \neg S25 \wedge \neg S26 \wedge \neg S1b \wedge \neg S15 \wedge \neg S2f$.

Table VIII shows the literals that are represented by the corresponding slices. An example attack characterized by pc_1 is `0 or 1/**/`, which is an obfuscated variation of a tautology attack, e.g. `'' or 1 = 1`. All attacks characterized by pc_1 contain the slices $S1d$, $Sf1$, and $S5$. If a WAF blocks input containing any of the three slices, the attacks characterized by pc_1 do no longer bypass.

TABLE X
PATH CONDITIONS FROM GENERATION 5

Id	Path condition
pc_2	$S1d \wedge S57 \wedge \neg S25 \wedge \neg S26 \wedge \neg Sc \wedge \neg S14 \wedge \neg S1 \wedge \neg S3e \wedge \neg S0 \wedge \neg S38 \wedge \neg S29 \wedge \neg S5 \wedge \neg Se1 \wedge \neg S1c \wedge \neg S1de \wedge \neg S76 \wedge \neg S67$
pc_3	$S1d \wedge S26 \wedge S15 \wedge \neg Sf \wedge \neg S1c \wedge \neg Se1 \wedge \neg S25 \wedge \neg S5 \wedge \neg Sc \wedge \neg S14 \wedge \neg S1de \wedge \neg S3e \wedge \neg S76 \wedge \neg S29 \wedge \neg S38 \wedge \neg S67$
pc_4	$S1d \wedge S29 \wedge \neg Se1 \wedge \neg S25 \wedge \neg S5 \wedge \neg S14 \wedge \neg S1de \wedge \neg S3e \wedge \neg S76 \wedge \neg S38 \wedge \neg S67$
pc_5	$S1d \wedge Sf \wedge S26 \wedge \neg S1c \wedge \neg Se1 \wedge \neg S25 \wedge \neg S5 \wedge \neg Sc \wedge \neg S14 \wedge \neg S1de \wedge \neg S3e \wedge \neg S76 \wedge \neg S29 \wedge \neg S38 \wedge \neg S67$
pc_6	$S1d \wedge S0 \wedge \neg S25 \wedge \neg S26 \wedge \neg Sc \wedge \neg S14 \wedge \neg S3e \wedge \neg S38 \wedge \neg S29 \wedge \neg S5 \wedge \neg Se1 \wedge \neg S1c \wedge \neg S1de \wedge \neg S76 \wedge \neg S67$
pc_7	$S1d \wedge Sc \wedge \neg S1c \wedge \neg Se1 \wedge \neg S25 \wedge \neg S5 \wedge \neg S14 \wedge \neg S1de \wedge \neg S3e \wedge \neg S76 \wedge \neg S29 \wedge \neg S38 \wedge \neg S67$
pc_8	$S1d \wedge S5 \wedge S26 \wedge S2 \wedge S0 \wedge Sb \wedge \neg Sa8 \wedge \neg S7c3 \wedge \neg Sf \wedge \neg S1e1 \wedge \neg S25 \wedge \neg Sd \wedge \neg Sc \wedge \neg S14 \wedge \neg S15 \wedge \neg S26d \wedge \neg S60 \wedge \neg S2e \wedge \neg S2d \wedge \neg S38 \wedge \neg S3$
pc_9	$S1d \wedge S3 \wedge S5 \wedge S6 \wedge \neg Sf \wedge \neg S25 \wedge \neg Sf1 \wedge \neg Sc \wedge \neg S26 \wedge \neg S1b \wedge \neg S5a \wedge \neg S14 \wedge \neg S71$

As shown by our quantitative analysis, the number of discovered path conditions increases during test execution. With more path conditions, the reasons why attacks are bypassing can be more precisely characterized. To illustrate this phenomenon, we analyze the path conditions obtained in different generations of the same test run. To limit the number of path conditions in this example, we only consider path conditions that contain the slice $S1d$, such as in pc_1. This slice represents the SQL keyword `or` and can, for example, be a part of an SQL tautology attack, e.g.,`'' or 1=1`.

The previously analyzed path condition pc_1 is obtained from the first generation of a test run. In contrast, Table X shows more path conditions from the same test run obtained after the fifth generations. Table IX depicts the string patterns that can be devised from the path conditions.

The first notable difference between both sets of path conditions is their number. In the first generation, only one path condition contains slice $S1d$ while there are eight of them by the fifth generation. When analyzing the patterns identified by the path conditions, there are several interesting observations. First, pc_2–pc_7 identify the patterns of bypassing attacks that use slice $S1d$, but require $S5$ to be absent ($\neg S5$), thus, suggesting the pattern identified in the first generation was incomplete. For example, pc_5 characterizes bypassing attacks containing slice $S1d$ combined with the Boolean identifier `true` and the comment symbol `#`, e.g., `'' or true#`. Such attacks are not matched by the pattern represented by pc_1 and, thus, would still bypass the WAF if only pc_1 would be considered in fixing the WAF.

To conclude, we see from the example above that having more path conditions helps derive a better understanding of the patterns shared by bypassing attacks. In turn, this puts a firewall administrator in a better position to devise an effective patch for a WAF's rule set, as we will demonstrate in Section VI.

VI. Using Attack Patterns to Repair a WAF

In this section, we demonstrate how the identified bypassing attacks and corresponding path conditions can be used to improve the WAF's rule set. We also discuss the benefits of `ML-Driven E` compared to the other `ML-Driven` strategies, RAN, SqlMap, and WAF Testing Framework.

A. Repair Strategy

Typically, a rule set uses regular expressions to match known malicious attacks (refer to Section II-A). Our goal is to modify these regular expressions in such a way that the bypassing attacks found by `ML-Driven` do no longer bypass the WAF. We investigated a first attempt to semiautomate the WAF repairing process in our recent work [11]. In particular, we use multiobjective optimization algorithms to optimize two goals: 1) maximizing the number of blocked attacks; and 2) minimizing the number of legitimate requests being blocked (false positives).

Algorithm 5: A Strategy to Repair a Faulty WAF Rule Set Using Bypassing Attacks and Path Conditions.

```
1:  Inputs:
        $A$: a set of bypassing attacks such that $|A| > 0$
        $PC$: a set of path conditions learned from $A$
        $R$: a set of regular expressions such that
            $\forall\, a \in A \nexists\, r \in R : r$ matches $a$
2:  Output:
        $R'$: a set of regular expressions such that
            $\forall\, a \in A \exists\, r \in R' : r$ matches $a$
3:  $R' \leftarrow R$
4:  while $PC \neq \emptyset$ do
5:      $s \leftarrow$ selectMostFrequentSlice($PC$)
6:      $PC_s \leftarrow$ getPathConditionsContainingSlice($s$)
7:      $A_s \leftarrow$ getCharacterisedAttacks($PC_s$)
8:      Select $r_i \in R$ that matches attacks similar to $A_s$
9:      Modify rule $r_i$ to $r_i^*$ such that it matches $A_s$
10:     $R' \leftarrow (R' \setminus \{r_i\}) \cup \{r_i^*\}$
11:     $PC \leftarrow PC \setminus PC_s$
12: end while
13: return $R'$
```

In this paper, we consider a general strategy for modifying the regular expressions as outlined in Algorithm 5. While parts of the strategy can be automated [11], regular expressions derived from attack patterns (line 2 and 2) still require manual inspection and adaptation depending on the WAF under evaluation. The required inputs for the strategy are the test outputs of `ML-Driven`, i.e., a set A of bypassing attacks and a set PC of path conditions learned from A, and a set R of regular expressions used by the WAF under test to match known attacks. Note that the attacks in A are not matched by any regular expression in R, otherwise the attacks would not bypass in the first place, and, hence, $\forall\, a \in A \nexists\, r \in R : r$ matches a. The output of the strategy is a set of regular expressions R', which results from modifying some regular expressions in R such that they match all attacks in A. We will use a running example in which A and PC are initialized with the attacks and path conditions from generation five of the previous examples (refer to Table IX). R is initialized with the regular expressions from the proprietary WAF used in our evaluation. Note that the examples are not artificial as the regular expressions were actively protecting the industrial web service application used in our evaluation. The attacks in Table IX would have bypassed the WAF deployed in the production system.

The strategy starts in line 2 by initializing the output variable R' by assigning R to it. The while loop from line 2 to 2 is repeated until all path conditions in PC are processed. In line 2, the loop begins by selecting a slice s that is the most frequently contained slice among the path conditions in PC. Line 2 selects all path conditions $PC_s \subset PC$ that contain slice s. Then, line 2 selects all attacks $A_s \subset A$ that are characterized by the path conditions in PC_s. In the running example, the slice $S1d$ is assigned to s because it is a part of eight path conditions (pc_2–pc_9) and, hence, is a part of more path conditions than any other slice. The path conditions pc_2–pc_9 (see Table X) are assigned to PC_s because they contain slice $S1d$. Accordingly, the attacks characterized by the path conditions pc_2–pc_9 are assigned to A_s (refer to Table IX).

In line 8, a security analyst manually inspects the regular expressions in R and identifies a regular expression r_i that should be modified in order to correctly identify the bypassing attacks in A_s. Depending on the regular expressions R, there might already exist an expression $r_i \in R$ that identifies attacks similar to those in A_s and, hence, could be modified to also account for A_s. Alternatively, if no such regular expression exists, a new regular expression can be created or, if there is more than one candidate regular expression for modification, several regular expressions can be modified so that each expression matches a subset of A_s. The security analyst is free to make this decision based on how he intends to logically structure the rule set. In the running example, we assign the following regular expression to r_i:

$$\texttt{(?i)'}[\backslash s]{+}\texttt{or}.$$

The regular expression begins with `(?i)`, which enables case-insensitive matching. It matches strings that contain an apostrophe followed by at least one whitespace ($[\backslash s]^+$) followed by the SQL keyword `or`. We select this expression for modification because it is the only expression in the rule set that tries to match attacks containing the SQL keyword `or`. Since all attacks in A_s also contain `or`, the expression is a suitable candidate for modification.

In line 9, the security analyst extends the regular expression r_i so that it also matches the attacks in A_s. The path conditions in PC_s provide guidance on how r_i could be extended. In the running example, when we contrast the expression r_i with the path conditions in PC_s, we make several observations regarding why r_i does not identify the attacks in A_s:

1) unlike in r_i, not all attacks in A_s start with an apostrophe, but they start either with an apostrophe, a double quote (e.g., pc_7), or a digit (e.g., pc_6);
2) unlike in r_i, none of the path conditions requires a leading whitespace before the literal `or`; and
3) r_i does not account for SQL comment characters, i.e., `#` and `/**/`, which are frequently used in bypassing attacks (e.g., pc_3, pc_5, pc_8, and pc_9).

Based on these observations, we devise an improved regular expression r_i^* to avoid the mentioned shortcomings:

$$\texttt{(?i)(('|"|\textbackslash d).*or)|(or.*(/**/|\#)).}$$

The modified expression r_i^* consists of two parts. The first part is `(('|"|\d).*or)` and addresses 1) as well as 2). It addresses 1) by replacing the apostrophe with `('|"|\d)` and it addresses 2) by replacing `[\s] +` with an arbitrary character sequence (denoted by `.*`). Therefore, the first part matches strings that contain either an apostrophe, a double quote, or a digit, followed by an arbitrary character sequence, and followed by `or`. The second part is `(or.*(/**/|#))` and addresses 3) as it matches strings that contain `or` and an SQL comment character (`/**/` or `#`). Since both parts are joined with an alternation (`|`), the strings matched by r_i^* is the union set of the strings matched by the first part and the second part.

With the performed changes, r_i^* matches all attacks in A_s. In line 2, we remove the old expression r_i from the output set R' and add r_i^* to R'. Since all attacks in A_s are now correctly identified, we remove in line 11 the path conditions PC_s from PC and continue with the next loop iteration until all path conditions in PC are processed. In the running example, PC is empty after the first iteration and the main loop ends.

Note that the proposed strategy helps a security expert to efficiently deal with the typically large number of path conditions found by `ML-Driven` by grouping conditions containing similar attack patterns. Grouping patterns helps the security experts to consider relevant patterns together when devising an improved regular expression, and hence, they do not have to consider each pattern individually. The strategy is defined in such a way that the most frequent attack patterns are tackled first.

B. *Suitability of Other Techniques for the Purpose of Repairing a WAF*

The testing techniques and tools discussed throughout this paper find bypassing attacks with varying effectiveness. This section discusses how suitable the different techniques are for the purpose of repairing a WAF.

In our experiments, SqlMap and the WAF testing framework did find none or very few bypassing attacks and, hence, do not provide much useful information for repairing a WAF as many bypassing attacks would remain uncovered. RAN is more effective in finding bypassing attacks than SqlMap and the WAF Testing Framework, but it does not identify common bypassing attack patterns. In absence of such patterns, a security analyst has to translate a possibly large number of bypassing attacks into an WAF patch. This limitation affects not only RAN but also the other tools (i.e., SqlMap and WAF Testing Framework) as they provide the set of bypassing attacks without any additional information about which substrings are associated with bypassing the WAF. Instead, our `ML-Driven` techniques provide both the set of bypassing attacks (which are larger in number) and the associated attack patterns identified by machine learning algorithms (i.e., RandomTree and RandomForest).

TABLE XI
COMPARISON OF THE ATTACK PATTERNS FOUND BY `ML-Driven E`, `ML-Driven B`, AND `ML-Driven D`

Id	Pattern	ML-E	ML-B	ML-D
pc_1	*or␣1, /**/*	√	√	√
pc_2	*or, ␣1*	√	√	✗
pc_3	*or,), #*	√	√	√
pc_4	*or, ␣0*	√	√	√
pc_5	*or, true, #*	√	✗	✗
pc_6	*or, 0*	√	√	√
pc_7	*or, ”*	√	√	√
pc_8	*or, /**/, #, ␣, 0, !*	√	✗	✗
pc_9	*or, /**/, 1, ~␣1*	√	√	✗

Amongst the `ML-Driven` techniques, `ML-Driven E` was found to be the most efficient. Having more successful bypassing attacks provide additional benefits for security analysts in charge of repairing the vulnerable WAF. First, a lower number of distinct attacks will lead to less attack patterns being identified and less new regular expressions to be added to the WAF rule set. Therefore, `ML-Driven` B/D may miss some distinct attacks, leading to less robust patches as the fixed WAF would not be able to detect/match the missed patterns. Second, machine learning algorithms better identify useful and useless attack patterns (or path conditions) when more bypassing attacks are provided as shown in Section V-D. For example, let us assume that `ML-Driven` B/D generate one single bypassing attack with the following slices: $S_1 =$ ', $S_2 =$ ␣, $S_3 =$ `"a"="a"`, $S_4 =$ #. With one single attack, it may be difficult for security analysts to determine which of these slices is actually associated with bypassing the WAF. Therefore, deeper analysis (or more attacks) is needed to determine which slices should form regular expressions to add to the WAF's rule set.

To illustrate the practical usefulness of the three `ML-Driven` techniques, let us now compare the generated attacks patterns for the running example used in Section VI-A. It corresponds to one of the regular expressions that were actively protecting the industrial web service application used in our evaluation. Table XI lists the attack patterns that were found in the same amount of time by `ML-Driven E`, `ML-Driven B` and `ML-Driven D`, respectively. As we can notice, `ML-Driven E` generated more attack patterns compared to the other `ML-Driven` variants: `ML-Driven E` misses pc_5 and pc_8 while `ML-Driven D` further misses pc_2 and pc_9. Assuming the analyst would rely on `ML-Driven B`, she would add the following regular expression containing all found attack patterns: $r = pc_1|pc_2|pc_3|pc_4|pc_6|pc_7|pc_9$, i.e., r matches the attacks satisfying either pc_1-pc_4, pc_6-pc_7, or pc_9. However, this patch would miss attacks found by `ML-Driven E` as the generated r would not match any attacks containing either pc_5 or pc_8. The patches generated with `ML-Driven D` would be less robust compared to `ML-Driven E` and `ML-Driven B` as the resulting regular expressions would not match attacks satisfying the missing four patterns.

The example above shows that generating more distinct attacks leads to uncovering more successful attack patterns and implementing more robust patches to the WAF.

VII. DISCUSSION

This section discusses some practical implications of our `ML-Driven` approach and its empirical results.

A. *Differences Between Case Studies*

When comparing the testing results of the two case studies, some notable differences stand out. First, bypassing attacks could be found for each parameter protected by ModSecurity, whereas with the proprietary WAF, only 29 out of 75 parameters lead to bypassing attacks. This can be attributed to the fact that the latter strictly validates each input to follow an expected format. For example, a value provided to the parameter *credit card number* must consist of 16 to 19 digits and, otherwise, the request is rejected. SQLi attacks typically require a larger character set, and thus, all attacks are blocked. Similarly, for the 46 parameters for which no attack is found, the expected input format prevents attacks. However, it is not possible to define such strict validation rules for all parameters, since the inputs might vary significantly in terms of character set and length. This is the case for the other 29 parameters where the expected input format is very general and the input validation rules rather loose, thus being prone to attacks. For example, the vulnerable parameter *Address* is expected to be a string with a maximum of 35 characters, a constraint with which many of the SQLi attacks comply.

Another major difference between the two case studies is the number of bypassing attacks per tested parameter. For ModSecurity, about 1000 bypassing attacks per parameter are found while, for the proprietary WAF, they are on average 10000. This significant difference can be attributed to the attack detection capabilities of each respective firewall and highlights the difficulty of customizing a rule set for a particular IT environment in practice. In our experiment, we use a default rule set for ModSecurity, while the proprietary WAF has a customized rule set to match a particular IT system. Such a customization is often necessary to achieve an acceptable false positive rate, but comes at the cost of reduced attack detection capabilities in part due to the lack of suitable tools to test the firewalls.

B. *Application of the Proposed Techniques*

We have proposed and evaluated three variants of a machine-learning-driven technique for the generation of SQLi attacks, namely, `ML-Driven E`, `ML-Driven D`, and `ML-Driven B`. `ML-Driven D` and `ML-Driven B` entail different strategies in allocating the test generation budget. `ML-Driven E` reconciles these differences and delivers a better performance. We have compared all these variants with RAN, the baseline technique considered in our work, on ModSecurity (a popular open-source WAF), and a proprietary WAF. Our experiments show that `ML-Driven E` outperforms all the other techniques.

We have also demonstrated the usefulness of mining more bypassing attacks in devising string patterns to fix WAFs. In our context, we experimented with RandomTree and RandomForest as machine classifiers for `ML-Driven E`. Since we show that the latter helps extract more path conditions that are useful in identifying patterns and fixing WAFs, we recommend the use of RandomForest.

VIII. Conclusion

WAFs play an important role to protect online systems. The rising occurrence of new kinds of attacks and their increasing sophistication require that firewalls be updated and tested regularly, as otherwise attacks might remain undetected and reach the systems under protection.

We propose `ML-Driven`, a search-based approach that combines machine learning and EAs to automatically test the attack detection capabilities of WAFs. The approach automatically generates a diverse set of attacks, sends them to a WAF under test, and checks if they are correctly identified. By incrementally learning from the tests that are blocked or bypassing the firewall, our approach selects tests that exhibit string patterns with high bypassing probabilities (according the machine learning) and mutates them using an attack grammar designed to generate new and hopefully successful attacks. Identified bypassing attacks can be used to learn path conditions, which characterize successful attack patterns.

With such a set of bypassing attacks and path conditions that characterize them, a security expert can fix or fine-tune the WAF rules in order to block imminent SQLi attacks. In the attacker–defender war, time is vital. Being able to quickly learn and anticipate more attacks that can circumvent a firewall, in a timely manner, is very important to secure business data and services.

Though our approach was applied to SQLi attacks in this paper, it can be adapted to other forms of attacks by making use of other attack grammars targeting different types of vulnerabilities.

Our key contributions in this paper include the following:

1) enhancing our preliminary techniques by consolidating them and improving their performance;
2) comparing two different and adequate machine learning classifiers;
3) carrying out a large-scale evaluation on two popular WAFs; and
4) comparing our approach with state-of-the-art tools.

Evaluation results suggest that the performance of `ML-Driven` (and its enhanced variant in particular) is effective at generating many undetected attacks and provides a good basis to identify attack patterns to protect against. Furthermost, it also fares significantly better than the best available tools.

In our future work, we will investigate automated approaches to generate effective patches for the WAF under test starting from the learned attack patterns. We reported on an initial attempt to automate the repairing process in a recent paper [11], where we generated patches that block as many bypassing attacks as possible while limiting the blocking of legitimate inputs. Investigating further, more effective, repairing strategies that better exploit the attack patterns generated by `ML-Driven E` is a part of our future agenda. Finally, we plan to investigate various strategies (e.g., data resampling strategies, weighted training, and penalty-based training) to improve the effectiveness and the efficiency of `ML-Driven` by addressing the data imbalance problem.

Acknowledgment

This work builds on Dennis Appelt's Ph.D. dissertation.

References

[1] A. D. Brucker, L. Brügger, P. Kearney, and B. Wolff, "Verified firewall policy transformations for test case generation," in *Proc. Third Int. Conf. Softw. Testing, Verification Validation*, 2010, pp. 345–354.

[2] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos, "Management of an academic HPC cluster: The UL experience," in *Proc. Int. Conf. High Performance Comput. Simul.*, Bologna, Italy, Jul. 2014, pp. 959–967.

[3] E. Al-Shaer, A. El-Atawy, and T. Samak, "Automated pseudo-live testing of firewall configuration enforcement," *IEEE J. Sel. Areas Commun.*, vol. 27, no. 3, pp. 302–314, Apr. 2009.

[4] S. Anand *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *J. Syst. Softw.*, vol. 86, no. 8, pp. 1978–2001, 2013.

[5] N. Antunes, N. Laranjeiro, M. Vieira, and H. Madeira, "Command injection vulnerability scanner for web services," 2009. [Online]. Available: http://eden.dei.uc.pt/ mvieira/

[6] N. Antunes, N. Laranjeiro, M. Vieira, and H. Madeira, "Effective detection of SQL/XPath injection vulnerabilities in web services," in *Proc. 6th IEEE Int. Conf. Services Comput.*, 2009, pp. 260–267.

[7] D. Appelt, N. Alshahwan, and L. Briand, "Assessing the impact of firewalls and database proxies on SQL injection testing," in *Proc. 1st Int. Workshop Future Internet Testing*, 2013, pp. 32–47.

[8] D. Appelt, A. Nguyen, Cu D. Panichella, and L. Briand, "Automated testing of web application firewalls," Univ. Luxembourg, Luxembourg City, Luxembourg, Tech. Rep. TR-SnT-2016-1, 2016.

[9] D. Appelt, C. D. Nguyen, and L. Briand, "Behind an application firewall, are we safe from sql injection attacks?" in *Proc. IEEE 8th Int. Conf. Softw. Testing, Verification Validation*, 2015, pp. 1–10.

[10] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan, "Automated testing for sql injection vulnerabilities: An input mutation approach," in *Proc. 2014 Int. Symp. Softw. Testing Anal.*, 2014, pp. 259–269.

[11] D. Appelt, A. Panichella, and L. C. Briand, "Automatically repairing web application firewalls based on successful SQL injection attacks," in *Proc. 28th IEEE Int. Symp. Softw. Rel. Eng.*, Toulouse, France, Oct. 23–26, 2017, pp. 339–350.

[12] 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*. San Francisco, CA, USA: Morgan Kaufmann, 1998.

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

[14] S. W. Boyd and A. D. Keromytis, "Sqlrand: Preventing sql injection attacks," in *Applied Cryptography and Network Security*. New York, NY, USA: Springer, 2004, pp. 292–302.

[15] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[16] J. Coffey, L. White, N. Wilde, and S. Simmons, "Locating software features in a SOA composite application," in *Proc. IEEE 8th Eur. Conf. Web Services*, 2010, pp. 99–106.

[17] L. Desmet, F. Piessens, W. Joosen, and P. Verbaeten, "Bridging the gap between web application firewalls and web applications," in *Proc. 4th ACM Workshop Formal Methods Security*, 2006, pp. 67–77.

[18] A. Doupé, M. Cova, and G. Vigna, "Why johnny canâĂŹt pentest: An analysis of black-box web vulnerability scanners," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2010, pp. 111–131.

[19] M. Felderer, M. BÃijchler, M. Johns, A. D. Brucker, R. Breu, and A. Pretschner, "Security testing," *Adv. Comput.*, vol. 101, pp. 1–51, 2016.

[20] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao, "A static analysis framework for detecting sql injection vulnerabilities," in *Proc. 31st Annu. Int. Comput. Softw. Appl. Conf.*, Jul. 2007, vol. 1, pp. 87–96.

[21] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," *ACM Sigplan Notices*, vol. 43, pp. 206–215, 2008.

[22] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, pp. 1–8, 2012.

[23] P. Godefroid *et al.*, "Automated whitebox fuzz testing," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2008, vol. 8, pp. 151–166.

[24] W. Halfond, J. Viegas, and A. Orso, "A classification of sql-injection attacks and countermeasures," in *Proc. IEEE Int. Symp. Secure Softw. Eng.*, 2006, vol. 1, pp. 13–15.

[25] W. G. Halfond, S. Anand, and A. Orso, "Precise interface identification to improve testing and analysis of web applications," in *Proc. 18th Int. Symp. Softw. Testing Anal.*, 2009, pp. 285–296.

[26] W. G. Halfond and A. Orso, "Amnesia: Analysis and monitoring for neutralizing sql-injection attacks," in *Proc. 20th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2005, pp. 174–183.

[27] W. G. J. Halfond and A. Orso, "Preventing SQL injection attacks using AMNESIA," in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 795–798.

[28] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explorations Newsl.*, vol. 11, no. 1, pp. 10–18, Nov. 2009.

[29] J. Hwang, T. Xie, F. Chen, and A. X. Liu, "Systematic structural testing of firewall policies," in *Proc. IEEE Symp. Rel. Distrib. Syst.*, 2008, pp. 105–114.

[30] J. Jürjens and G. Wimmel, "Specification-based testing of firewalls," in *Perspectives of System Informatics* (Lecture Notes in Computer Science, vol. 2244), D. Bjørner, M. Broy, and A. Zamulin, Eds. Berlin, Germany: Springer, 2001, pp. 308–316.

[31] G. Karafotias, M. Hoogendoorn, and A. E. Eiben, "Parameter control in evolutionary algorithms: Trends and challenges," *IEEE Trans. Evol. Comput.*, vol. 19, no. 2, pp. 167–187, Apr. 2015.

[32] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of SQL injection and cross-site scripting attacks," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 199–209.

[33] Y.-F. Li, P. K. Das, and D. L. Dowe, "Two decades of web application testing: A survey of recent advances," *Inf. Syst.*, vol. 43, pp. 20–54, 2014.

[34] A. Liu, Y. Yuan, D. Wijesekera, and A. Stavrou, "Sqlprob: A proxy-based architecture towards preventing sql injection attacks," in *Proc. 2009 ACM Symp. Appl. Comput.*, 2009, pp. 2054–2061.

[35] H. Liu and H. B. Kuan Tan, "Testing input validation in web applications through automated model recovery," *J. Syst. Softw.*, vol. 81, no. 2, pp. 222–233, 2008.

[36] P. McMinn, "Search-based software test data generation: A survey," *Softw. Testing, Verification Rel.*, vol. 14, no. 2, pp. 105–156, 2004.

[37] R. McNally, K. Yiu, D. Grove, and D. Gerhardy, "Fuzzing: The state of the art," DTIC Doc., Tech. Rep. DSTO–TN–1043, 2012.

[38] J. Offutt, Y. Wu, X. Du, and H. Huang, "Bypass testing of web applications," in *Proc. 15th Int. Symp. Softw. Rel. Eng.*, 2004, pp. 187–197.

[39] A. Panichella, F. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Trans. Softw. Eng.*, vol. 44, no. 2, pp. 122–158, Feb. 2018.

[40] A. Panichella, F. M. Kifetew, and P. Tonella, "Reformulating branch coverage as a many-objective optimization problem," in *Proc. 8th IEEE Int. Conf. Softw. Testing, Verification Validation*, Graz, Austria, Apr. 13–17, 2015, pp. 1–10.

[41] A. Petrowski and S. Ben-Hamida, *Evolutionary Algorithms*. New York, NY, USA: Wiley, 2017.

[42] J. R. Quinlan, *C4.5: Programs for Machine Learning*, vol. 1. San Mateo, CA, USA: Morgan Kaufmann, 1993.

[43] D. Senn, D. Basin, and G. Caronni, "Firewall conformance testing," in *Testing of Communicating Systems* (Lecture Notes in Computer Science, vol. 3502), F. Khendek and R. Dssouli, Eds. Berlin, Germany: Springer, 2005, pp. 226–241.

[44] S. Shamshiri, J. M. Rojas, G. Fraser, and P. McMinn, "Random or genetic algorithm search for object-oriented test suite generation?" in *Proc. Annu. Conf. Genet. Evol. Comput.*, 2015, pp. 1367–1374.

[45] L. K. Shar and H. B. K. Tan, "Defeating sql injection," *Computer*, vol. 3, pp. 69–77, 2013.

[46] M. Soltani, A. Panichella, and A. van Deursen, "A guided genetic algorithm for automated crash reproduction," in *Proc. 39th Int. Conf. Softw. Eng.*, Buenos Aires, Argentina, May 20–28, 2017, pp. 209–220.

[47] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. London, U.K.: Pearson, 2007.

[48] O. Tripp, O. Weisman, and L. Guy, "Finding your way in the testing jungle: A learning approach to web security testing," in *Proc. Int. Symp. Softw. Testing Anal.*, 2013, pp. 347–357.

[49] M. Črepinšek, S.-H. Liu, and M. Mernik, "Exploration and exploitation in evolutionary algorithms: A survey," *ACM Comput. Surveys*, vol. 45, pp. 35, 2013.

[50] J. Williams and D. Wichers, "Owasp, top 10, the ten most critical web application security risks," Open Web Appl. Security Project, Tech. Rep., 2013. [Online]. Available: http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202010.pdf.

[51] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. San Mateo, CA, USA: Morgan Kaufmann, 2011.

Dennis Appelt received the Ph.D. degree in software engineering from the University of Luxembourg, Luxembourg City, Luxembourg, in 2016. His thesis was entitled "Automated security testing of web-based systems against SQL injection attacks."

He conducted his Ph.D. studies from July 2012 to June 2016 with the Interdisciplinary Centre for Security, Reliability, and Trust (SnT), University of Luxembourg, under the supervision of Prof. L. Briand. From July 2016 to November 2016, he was a Postdoctoral Researcher with SnT. He disseminated his Ph.D. work in the proceedings of various international conferences (e.g., ASE, ISSTA, ICST). He is currently a Software Engineer with the Research Department, Ripple, Inc., San Francisco, CA, USA, and works on secure protocol design and secure implementations for financial software. His research interests include software security, automated software testing, and distributed ledger technologies.

Cu D. Nguyen received the Ph.D. degree with a distinguished dissertation in computer science from the University of Trento, Trento, Italy, in 2009.

Before joining POST Luxembourg as a Data Scientist and Security Expert, he was a Researcher with the University of Luxembourg and a Postdoctoral Researcher with Fondazione Bruno Kessler, Trento, Italy. His fields of interests include machine learning, computer security, and secure software engineering.

Annibale Panichella received the Ph.D. degree in software engineering from the University of Salerno, Fisciano, Italy, in 2014. His dissertation was entitled "Search-based software maintenance and testing."

He is an Assistant Professor with the Delft University of Technology, Delft, The Netherlands. He is also a Research Fellow with the Interdisciplinary Centre for Security, Reliability and Trust , University of Luxembourg, Luxembourg City, Luxembourg, where he was a Research Associate until January 2018. He serves and has served as a program committee member of various international conference (e.g., ICSE, GECCO, ICST, and ICPC) and as a reviewer for various international journals (e.g., TSE, TOSEM, TEVC, EMSE, STVR) in the fields of software engineering and evolutionary computation. His research interests include security testing, evolutionary testing, search-based software engineering, textual analysis, and empirical software engineering.

Lionel C. Briand (F'10) is a Professor of software verification and validation with the SnT Centre for Security, Reliability, and Trust, University of Luxembourg, Luxembourg City, Luxembourg, where he is also the Vice Director of the centre. He is currently running multiple collaborative research projects with companies in the automotive, satellite, financial, and legal domains. He has held various engineering, academic, and leading positions in five other countries. He became an IEEE Fellow in 2010 for his work on the testing of object-oriented systems. His research interests include software testing and verification, model-driven software development, search-based software engineering, and empirical software engineering.

Prof. Briand was a recipient of the IEEE Computer Society Harlan Mills Award and the IEEE Reliability Society Engineer-of-the-Year Award for his work on model-based verification and testing, respectively, in 2012 and 2013. He received an ERC Advanced grant in 2016—on the topic of modeling and testing cyber-physical systems—which is the most prestigious individual research grant in the European Union.