WAF-A-MoLE: Evading Web Application Firewalls through Adversarial Machine Learning

Luca Demetrio
luca.demetrio@dibris.unige.it
Università di Genova

Andrea Valenza
andrea.valenza@dibris.unige.it
Università di Genova

Gabriele Costa
gabriele.costa@imtlucca.it
IMT School for Advanced Studies Lucca

Giovanni Lagorio
giovanni.lagorio@unige.it
Università di Genova

ABSTRACT

Web Application Firewalls are widely used in production environments to mitigate security threats like SQL injections. Many industrial products rely on signature-based techniques, but machine learning approaches are becoming more and more popular. The main goal of an adversary is to craft *semantically* malicious payloads to bypass the *syntactic* analysis performed by a WAF.

In this paper, we present WAF-A-MoLE, a tool that models the presence of an adversary. This tool leverages on a set of mutation operators that alter the syntax of a payload without affecting the original semantics. We evaluate the performance of the tool against existing WAFs, that we trained using our publicly available SQL query dataset. We show that WAF-A-MoLE bypasses all the considered machine learning based WAFs.

CCS CONCEPTS

• **Security and privacy → Web application security**.

KEYWORDS

web application firewall, adversarial machine learning, sql injection, mutational fuzzing

ACM Reference Format:
Luca Demetrio, Andrea Valenza, Gabriele Costa, and Giovanni Lagorio. 2020. WAF-A-MoLE: Evading Web Application Firewalls through Adversarial Machine Learning. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3341105.3373962

1 INTRODUCTION

Most security breaches occur due to the exploitation of some vulnerabilities. Ideally, the best way to improve the security of a system is to detect all its vulnerabilities and patch them. Unfortunately, this is rarely feasible due to the extreme complexity of real systems and high costs of a thorough assessment. In many contexts, payloads arriving from the Internet are the primary threat, with the attacker using them to discover and exploit some existing vulnerabilities. Thus, protecting a system against malicious payloads is crucial. Common protection mechanisms include input filtering, sanitization, and other domain-specific techniques, e.g., *prepared statements*. Implementing effective input policies is non trivial and, sometimes, even infeasible (e.g., when a system must be integrated in many heterogeneous contexts).

```
admin' OR 1=1#
admin' OR 0X1=1 or 0x726!=0x726 OR 0x1Dd
  not IN/*(seleCt 0X0)>c^Bj>N]*/ ((SeLeCT
   476),(SELECT (SElEct 477)),0X1de) oR
  8308 noT lIkE  8308\x0c AnD truE OR '
  FZ6/q' LiKE 'fz6/qI' anD TRUE anD '>U'
  != '>uz'#t'%'03;Nd
```

Figure 1: Two semantically equivalent payloads.

For this reason, mitigation solutions are often put in place. For instance, *Intrusion Detection Systems* (IDS) aim to detect suspicious activities. Clearly, these mechanisms have no effect on existing vulnerabilities that silently persist in the system. However, when IDSs can precisely identify intrusion attempts, they significantly reduce the overall damage. The very core of any IDS is its detection algorithm: the overall effectiveness only depends on whether it can discriminate between harmful and harmless packets/flows.

Web Application Firewalls (WAFs) are a prominent family of IDS, widely adopted [16] to protect ICT infrastructures. Their detection algorithm applies to HTTP requests, where they look for possible exploitation patterns, e.g., payloads carrying a SQL injection. Since WAFs work at application-level, they have to deal with highly expressive languages such as SQL and HTML. Clearly, this exacerbates the detection problem.

To clarify this aspect, consider a classical SQL injection scenario where the attacker crafts a malicious payload x such that the query `SELECT * FROM users WHERE name='`x`' AND pw='`y`'` always succeeds (independently from y). Figure 1 shows two instances of such a payload. Notice that the two payloads are semantically equivalent. As a matter of fact, both reduce the above query to `SELECT * FROM users WHERE name='admin' OR` $\top$ `#`... where $\top$ is a tautology and . . . is a trail of commented characters. Ideally a WAF should reject both these payloads. However, when classification is based on a mere syntactical analysis, this might not happen. Hence, the goal of an attacker amounts to looking for some malicious payload that

SAC '20, March 30-April 3, 2020, Brno, Czech Republic

ACM ISBN 978-1-4503-6866-7/20/03. . . $15.00
https://doi.org/10.1145/3341105.3373962

is undetected by the WAF. We present a technique to effectively and efficiently generate such malicious payloads, that bypass ML-based WAF. Our approach starts from a target malicious payload that the WAF correctly detects. Then, by iteratively applying a set of mutation operators, we generate new payloads. Since mutation operators are semantics-preserving, the new payloads are equivalent from the point of view of the adversary. However, they gradually reduce the confidence of the WAF classification algorithm. Eventually, this process converges to a payload classified below the rejection threshold. To evaluate the effectiveness of our methodology we implemented a working prototype, called WAF-A-MoLE. Then we applied WAF-A-MoLE to different ML-based WAFs, and evaluated their robustness against our technique.

Contributions of the paper. The main contributions of this work are summarized as follows: (*i*) we develop a tool for producing adversarial examples against WAFs by leveraging on a set of syntactical mutations, (*ii*) we produce a dataset of both sane and injection queries, (*iii*) and we review the state of the art of machine learning SQL injection classifiers and we bypass them using WAF-A-MoLE.

2 PRELIMINARIES

Web Application Firewalls (WAFs) are commonly used to prevent application-level exploits of web applications. Intuitively, the idea is that a WAF can detect and drop dangerous HTTP requests to mitigate potential vulnerabilities of web applications. The most common detection mechanisms include *signature-based matching* and *classification via machine learning.*

Signature-based WAFs identify a payload according to a list of rules, typically written by some developers or maintained by a community. For instance, rules can be encoded through some policy specification language that defines the syntax of legal/illegal payloads. Nowadays, the signature-based approach is widely used and, perhaps, the most popular signature-based WAF is ModSecurity[1].

However, recently the machine learning-based approach has received increasing attention. For instance, both FortiNet[2] and PaloAlto[3] include ML-based detection in their WAF products, since ML can overcome some limitations of signature-based WAFs, i.e., the extreme complexity of developing a list of syntactic rules that precisely characterizes malicious payloads. Since ML WAFs are trained on existing legal and illegal payloads, their configuration is almost automatic.

Adversarial machine learning (AML) [6, 22] studies the threats posed by an attacker aiming to mislead machine learning algorithms. More specifically, here we are interested in *evasion attacks*, where the adversary crafts malicious payloads that are wrongly classified by the victim learning algorithm. The adversarial strategy varies with the target ML algorithm. Many existing systems have been shown to be vulnerable and several authors, e.g. [7, 10, 18, 31], proposed techniques for systematically generating malicious samples. Intuitively, the crafting process works by introducing a *semantics-preserving* perturbation in the payload, that interferes with the classification algorithms. Notice that, often, a formal semantics of the classification domain is not available, e.g., it is informally provided through an oracle such as a human classifier. The objective of the adversary may be written as a constrained minimization problem $x^* = \arg\min_{x, C(x)} \mathcal{D}(f(x), c_t)$, where f is the victim classifier, c_t is the desired class the adversary wants to reach, $\mathcal{D}$ is a distance function, and $C(x)$ represents all the constraints that cannot be violated during the search for adversarial examples. Since we consider binary classifiers, we can rewrite our problem as $x^* = \arg\min_{x, C(x)} f(x)$, where the output of f is bounded between 0 and 1, and we are interested in reaching the benign class represented by 0.

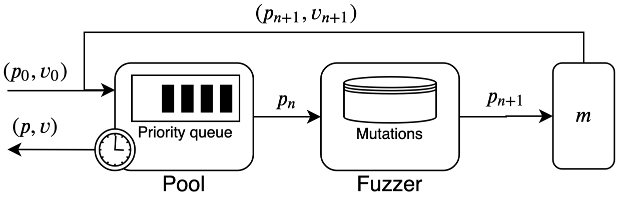

Figure 2: An outline of the mutational fuzz testing approach.

```
input: Model $m$, Payload $p_0$, Threshold $t$
output: head($Q$)

$Q$ := create_priority_queue()
$v$ := classify($m$, $p_0$)
enqueue($Q$, $p_0$, $v$)
while $v > t$
  $p$ := mutate(head($Q$))
  $v$ := classify($m$, $p$)
  enqueue($Q$, $p$, $v$)
```

Figure 3: Core algorithm of WAF-A-MoLE.

3 OVERVIEW OF WAF-A-MOLE

Our methodology belongs to the class of *guided mutational fuzz testing* approaches [17, 38]. Briefly, the idea is to start from a failing test, that gets repeatedly transformed through the random application of some predefined mutation operators. The modified tests, called *mutants*, are then executed, compared (according to some performance metric) and ordered. Then, the process is iterated on the tests that performed better until a successful test is found. Clearly, this approach requires both a comparison criterion and a set of mutation operators. These are typically application-dependent. Figure 2 schematically depicts this approach.

3.1 Algorithm description

In our context a test is a SQL injection and its execution amounts to submitting it to the target WAF. The comparison is based on the confidence value generated by the detection algorithm of the WAF. The payload pool is the data structure containing the SQL injection candidates to be mutated during the next round. Below we describe in more detail the set of mutation operators and the payload pool.

[1] https://modsecurity.org
[2] https://www.fortinet.com/blog/business-and-technology/fortiweb-release-6-0--ai-based-machine-learning-for-advanced-thr.html
[3] https://www.paloaltonetworks.com/detection-response

Operator	Short definition	Example
Case Swapping	$CS(\dots a \dots B \dots) \rightarrow \dots A \dots b \dots$	CS(`admin' OR 1=1#`) → `ADmIn' oR 1=1#`
Whitespace Substitution	$WS(\dots k_1 k_2 \dots) \rightarrow \dots k_1 \text{␣} k_2 \dots$	WS(`admin' OR 1=1#`) → `admin'\n OR \t 1=1#`
Comment Injection	$CI(\dots k_1 k_2 \dots) \rightarrow \dots k_1 \texttt{/**/} k_2 \dots$	CI(`admin' OR 1=1#`) → `admin'/**/OR 1=1#`
Comment Rewriting	$CR(\dots \texttt{/*} s_0 \texttt{*/} \dots \texttt{\#} s_1) \rightarrow \dots \texttt{/*} s'_0 \texttt{*/} \dots \texttt{\#} s'_1$	CR(`admin'/**/OR 1=1#`) → `admin'/*abc*/OR 1=1#xyz`
Integer Encoding	$IE(\dots n \dots) \rightarrow \dots 0x[n]_{16}$	IE(`admin' OR 1=1#`) → `admin' OR 0x1=1#`
Operator Swapping	$OS(\dots \oplus \dots) \rightarrow \dots \boxplus \dots$ (with $\oplus \equiv \boxplus$)	OS(`admin' OR 1=1#`) → `admin' OR 1 LIKE 1#`
Logical Invariant	$LI(\dots e \dots) \rightarrow \dots e \texttt{ AND } \top \dots$	LI(`admin' OR 1=1#`) → `admin' OR 1=1 AND 2<>3#`

Table 1: List of mutation operators.

A pseudo code implementation of the core algorithm of WAF-A-MoLE is shown in Figure 3. The algorithm takes the learning model $m : \mathcal{X} \rightarrow [0, 1]$, where $\mathcal{X}$ is the feature space, an initial payload p_0 and a threshold t, i.e., a confidence value under which a payload is considered harmless. WAF-A-MoLE implements the payload pool (see Section 3.3) as a priority queue Q (line 1). The payloads in Q are prioritized according to the confidence value returned by the classification algorithm, namely **classify**, associated to m. The classification algorithm assigns to each payload an $x \in \mathcal{X}$, by extracting a feature vector, and computes $m(x)$.

Initially, Q only contains p_0 (lines 2–3). The main loop (lines 4–7) behaves as follows. The head element of Q, i.e., the payload having the lowest confidence score, is extracted and mutated (line 5), by applying a set of mutation operators (see Section 3.2). The obtained payload, p, is finally classified (line 6) and en-queued (lines 7). The termination of the algorithm occurs when a p receives a score less or equal to the threshold t (line 4).

3.2 Mutation operators

A mutation operator is a function that changes the syntax of a payload so that the semantics of the injected queries is preserved.

Below we describe the considered mutation operators.

CS. The *Case Swapping* operator randomly changes the capitalization of the keywords in a query (e.g., `Select` to `sELecT`). Since SQL is case insensitive, the semantics of the query is not affected.

WS. *Whitespace Substitution* relies on the equivalence between several alternative characters that only act as separators (whitespaces) between the query tokens. For instance, whitespaces include `\n` (line feed), `\r` (carriage return) and `\t` (horizontal tab). Each of these characters can be replaced by an arbitrary, non-empty sequence of the others without altering the semantics of the query.

IC. Inline comments (`/*...*/`) can be arbitrarily inserted between the tokens of a query. Since comments are not interpreted, they are semantics preserving. The *Comment Injection* operator randomly adds inline comments between the tokens.

CR. Following the above reasoning, the *Comment Rewriting* operator randomly modifies the content of a comment.

IE. The *Integer Encoding* operator modifies the representation of numerical constants. This includes alternative base representations, e.g., from decimal to hexadecimal, as well as statement nesting, e.g., `(SELECT 42)` is equivalent to `42`.

OS. Some operators can be replaced by others that behave in the same way. For instance, the behavior of `=` (equality check) can be simulated by `LIKE` (pattern matching). We call this mutation *Operator Swapping*.

LI. A *Logical Invariant* operator modifies a boolean expression by *adding opaque predicates.*[4]

Table 1 provides a compact list, including a short, mnemonic definition, of the operators described above.

3.3 Mutation tree

The priority queue of Figure 3 contains a sequential representation of mutation tree. Starting from a root element, i.e., the initial payload (p_0 in Figure 3), a mutation tree contains elements obtained through the application of some mutation operator. A possible instance of a mutation tree is shown in Figure 4. Each edge is labeled with an identifier of the applied mutation operator. Also, each node is labeled with a possible classification value (in percentage). The corresponding queue is given by the sequence of the nodes in the mutation tree ordered by the associated classification value.

After applying a mutation (actually after a full *mutation round*, see Section 3.4), the payload is evaluated and added to the priority queue, along with information about the payload that generated it. Keeping all individuals in the initial population helps avoiding local minima: when a payload is unable to create better payloads, the algorithm tries to backtrack on old payloads to create a new branch on the mutation tree.

3.4 Efficiency

The main bottleneck of our algorithm is the classification step. Indeed, the classification of a payload requires the extraction of a vector of features. Although a WAF classifier is efficient, the feature extraction process may require non-negligible string parsing operations (see Section 4). For example, the procedure carried out by a token-based classifier (see Section 4.2 for details) requires non-trivial computation to parse the SQL query language (being context free). Instead, all the mutation operators described in Section 3.2 rely on efficient string parsing, based on regular expressions.

We mitigate this issue by following a *mutation preemption* strategy, i.e., we create a *mutation round* where multiple payloads are generated at once. All these mutated payloads are stored for the classification. Then we run all the classification steps in parallel and we discharge the mutants that increase the classification value of their parent. In this way we take advantage of the parallelization support of modern CPUs.

[4] That is, heuristically generated true and false expressions to be combined in conjunction and disjunction (respectively) with the payload clauses.

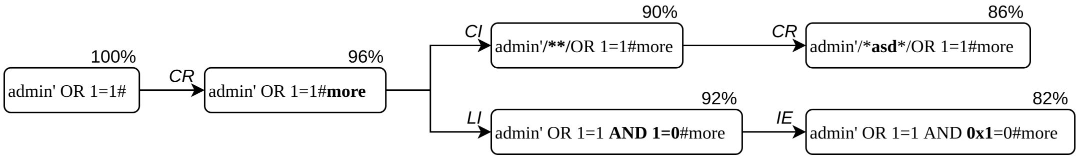

Figure 4: A possible mutation tree of an initial payload.

For memory efficiency, we only enqueue a mutated payload if it improves the classification value of its parent. In this way we mitigate the potential, exponential blow-up of the mutation tree (see Section 3.3). On the negative side, each branch of the mutation tree only evolves monotonically which might result in the algorithm stagnating on local minima. However, our experiments show that this does not prevent our algorithm from finding an injectable payload (see Section 5).

4 WAF TRAINING AND BENCHMARKING

Our technique applies to an input model representing a *well-trained* WAF, i.e., a WAF that effectively detects malicious payloads. Ideally, to generate a payload that bypasses a deployed WAF, the input algorithm should rely on the same detection model. In the case of ML-based WAF, the model is the result of the training process over a sample dataset, while for signature-based WAFs the model is the set of all the collected signatures that are used as a comparison for future input data.

Unfortunately, it is very common that neither the detection model nor the training dataset are publicly available. Reasonably, this happens because the WAF manufacturers (correctly) consider such knowledge an advantage for the adversary. Remarkably, this also happens for the research prototypes.[5] Thus, we had to create a training dataset and configure the classification algorithms. The following sections describe the issues we faced during this process and how we solved them.

4.1 Dataset

To the best of our knowledge, no dataset of benign SQL queries is publicly available. The main reason is probably that the notion of "benign" is application-dependent and no universal definition exists. On the other hand, there are many malicious payloads, that one can extract from existing penetration testing tools such as *sqlmap*[6] and *OWASP ZAP*[7]. We consider the payloads generated by these tools, as any WAF should be trained on well-known attacks.

We built our dataset through an automatic procedure[8]. In particular, we used *randgen*[9] to generate the queries. Starting from a grammar G, the tool returns a set of queries that belong to the language denoted by G. Noticeably, queries generated by *randgen* also include actual values, e.g., table and column names, referring to a given existing database. Thus, the queries in the dataset can be submitted and evaluated against a real target.

To create our labeled dataset, we assume that SQL queries are always created by the application when a user submits a payload, either benign or malicious. To simulate this behavior, we generate a single initial grammar that supports multiple query types. Then, we provide different dictionaries of values for each terminal symbol (i.e., t, f, v) that represents a possible value of a particular column inside the database.

The query grammar is the following.

$$\begin{array}{lcl}
Q & ::= & S \mid U \mid D \mid I \\
S & ::= & \textbf{SELECT } (\bar{f} \mid *) \textbf{ FROM } t \textbf{ WHERE } e \ [\textbf{LIMIT } \bar{v}] \\
U & ::= & \textbf{UPDATE } t \textbf{ SET } f = v \textbf{ WHERE } e \ [\textbf{LIMIT } \bar{v}] \\
D & ::= & \textbf{DELETE FROM } t \textbf{ WHERE } e \ [\textbf{LIMIT } \bar{v}] \\
I & ::= & \textbf{INSERT INTO } t \ (\bar{f}) \textbf{ VALUES } (\bar{v}) \\
e & ::= & f \gtreqless v \mid f \textbf{ LIKE } s \mid e \textbf{ AND } e' \mid e \textbf{ OR } e'
\end{array}$$

Briefly, the queries Q can be *select* S, *update* U, *delete* D or *insert* I. The syntax of each query is standard, only notice that S, U and D may optionally (square brackets) terminate with a **LIMIT** clause. The queries operate on several parameter types, including fields f, tables t, values v, strings s and boolean expressions e, e'. Finally, we use $\bar{\cdot}$ to denote a vector, i.e., a finite, comma-separated list of elements. The actual values for t and f are taken from an actual target database (this feature is provided by *randgen*). For v, we use different values depending on the type of query we want to generate. For the benign queries, we generate payloads with a random generator, a dictionary of nations, a dictionary of values which are compatible with the field type to simulate a real application payload. For example, in a database containing people names we use English first and last names. We are interested in the structure of the query, hence these values for the payload are suitable for our analysis.

As mentioned above, the malicious values are generated by *sqlmap* and *OWASP ZAP*.

4.2 Classification algorithms

Below we describe the classification algorithms that we used for our experiments. In particular, we consider different techniques, built on three feature extraction methods: characters, token and graph based.

Character-based features. WAF-Brain[10] is based on a recurrent-neural network. The network divides the input query in blocks of exactly five consecutive characters. Its goal is to predict the sixth character of the sequence based on the previous five. If the prediction is correct, the block of characters is more likely to be part of a malicious payload. This process is repeated for every block of five characters forming the target query.

[5] All maintainers of the WAFs considered in this work were contacted, but no one provided their datasets.
[6] https://github.com/sqlmapproject/sqlmap
[7] https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project
[8] The dataset is available at https://github.com/blindusername/wafamole-dataset
[9] https://github.com/MariaDB/randgen
[10] https://github.com/BBVA/waf-brain

		C	γ	$avg(A)$	σ
Token-based	Naive Bayes	/	/	54.2%	1.0%
	Random Forest	/	/	87.3%	0.7%
	Linear SVM	19.30	/	80.5%	1.4%
	Gaussian SVM	278.25	0.013	93.1%	0.9%
SQLiGoT	Dir. Prop.	4.64	0.26	99.85%	0.07%
	Undir. Prop.	2.15	0.71	99.10%	0.2%
	Dir. Unprop.	2.15	0.26	99.74%	0.1%
	Undir. Unprop.	2.15	0.26	98.89%	0.2%

Table 2: Training phase results.

		A	**R**	**P**
ModSecurity CSR	Paranoia 1/2	86.10%	86.10%	100%
	Paranoia 3/4	91.85%	91.85%	100%
	Paranoia 5	96.46%	96.46%	100%
WAF-Brain	RNN	98.27%	96.73	99.8%
Token-based	Naive Bayes	50.16%	98.71%	50.08%
	Random forest	98.33%	98.33%	100%
	Linear SVM	98.75%	98.76%	100%
	Gaussian SVM	97.82%	97.82%	100%
SQLiGoT	Dir. Prop.	90.61%	97.30%	85.82%
	Undir. Prop.	96.38%	97.31%	95.54%
	Dir. Unprop.	90.52%	97.12%	85.80%
	Undir. Unprop.	96.25%	97.05%	95.53%

Table 3: Benchmark table.

The neural network of WAF-Brain is structured as follows. The input layer is a Gated Recurrent Unit (GRU) [12] made of five neurons, followed by two fully-connected layers, i.e., a dropout layer followed by another fully connected layer. Finally, WAF-Brain computes the average of all the prediction errors over the input query and scores it as malicious if the result is above a fixed threshold chosen a priori by the user. Since the threshold is not given by the classifier itself, as all the other details of the training and cross-validation phases, we set it to 0.5, which is the standard threshold for classification tasks.

Token-based features. The token-based classifiers represent input queries as histograms of symbols, namely tokens. A token is a portion of the input that corresponds to some syntactic group, e.g., a keyword, comparison operators or literal values.

We took inspiration from the review written by Komiya et al. [27] and Joshi et al [24] and we developed a tokenizer for producing the features vector to be used by these models. On top of that, we implemented different models: (i) a Naive Bayes (NB) classifier, (ii) a random forest (RF) classifier with an ensemble of 25 trees, (iii) a linear SVM (L-SVM), and a gaussian SVM (G-SVM). We trained them using a 5-fold cross-validation with 20,000 sane queries and 20,000 injections, and we used 15% of the queries for the validation set. To this extent, we coded our experiment using *scikit-learn* [32], which is a Python library containing already implemented machine learning algorithms. After the feature extraction phase, the number of samples dropped to 768 benign and 7,963 injection queries. The tokenization method is basically an aggregation method: only a subset of all symbols are taken into account. The dataset is unbalanced, as the variety of sane queries is outnumbered by the variety of SQL injections. To address this issue, we set up *scikit-learn* accordingly, by using a loss function that takes into account the class imbalance [9]. Table 2 shows the results of the training phase where (i) C is the regularization parameter [36] that controls the stability of the solution, (ii) γ is the kernel parameter (only for the gaussian SVM) [1, 21], and (iii) $avg(A)$ and σ are the average and standard deviation of the accuracy computed during the cross-validation phase over the validation set.

Graph-based features. Kar et al. [25] developed SQLiGoT, an SQL injection detector that represents a SQL query as a graph, both directed and undirected. Each node in this graph is a token of the SQL language, plus all system reserved and user defined table names, variables, procedures, views and column names. Moreover, the edges are weighted uniformly or proportionally to the distances in terms of adjacency. We omit all the details of the model, as they are well described in the paper. Kar et al. released the hyper-parameter they found on their dataset, but since both C and γ depend on data, we had to train these models from scratch.

We performed a 10-fold cross-validation for SQLiGoT, using 20,000 benign and 20,000 malicious queries, again using the *scikit-learn* library. After the feature extraction phase, the dataset is shrunk to: (i) 3216 sane 12,659 and malicious data for the directed graph versions, (ii) and 3268 sane 12,682 and malicious data for the undirected graph versions of SQLiGoT. Again, many queries possess the same structure as others, and this is likely to happen for sane queries. As already said in the previous paragraph, we are dealing with imbalance between the two classes, and we treat this issue by using a balanced accuracy loss function, provided by the *scikit-learn* framework. Table 2 shows the result of the training phase of the different SQLiGoT classifiers. Both the hyper-parameters and the scores are almost the same for all the different versions of SQLiGoT.

4.3 Benchmark

We carried out benchmark experiments to assess the detection rates of the classifiers discussed above. For all the classifiers used for this benchmark, we formed a dataset of 8,000 sane queries and 8,000 SQL injection queries, and we classified them using the models we have trained. Table 3 shows the results of our experiment.

We evaluated the performance of each classifier by accounting three different metrics: (i) *accuracy*, (ii) *recall*, and (iii) *precision*. We denote the true positives as TP, true negatives as TN, false positives as FP and false negatives as FN. Accuracy is computed as $A = \frac{TP+TN}{TP+TN+FP+FN}$, recall is computed as $R = \frac{TP}{TP+FN}$ and precision is computed as $P = \frac{TP}{TP+FP}$. The accuracy measures how many samples have been correctly classified, i.e., a sane query classified as sane or an injected query classified as malicious. The recall measures how good the classifier is at identifying samples from the relevant class, in this case the injection payloads. Scoring a high recall value means that the classifier labeled most of the real positives in the dataset as positives. The precision measures how many of the samples classified as relevant are actually relevant.

Since the Naive Bayes algorithm tries to discriminate between input classes by considering each variable independent one to another, it misses the real structure of the SQL syntax. Hence, it cannot properly capture the complexity of the problem. All other classifiers may be compared with different levels of paranoia offered by ModSecurity, showing their effectiveness as WAFs. WAF-Brain results are comparable to what the author claims on his GitHub repository.

5 EVADING MACHINE LEARNING WAFS

In this section, we experimentally assess WAF-A-MoLE against the classifiers introduced above. The experiments were performed on a DigitalOcean[11] droplet VM with 6 CPUs and 16GB of RAM. For a baseline comparison we used an *unguided* mutational fuzzer. The unguided fuzzer randomly applies the mutation operators of Section 3.2. Moreover, we executed 100 instances of the unguided fuzzer on each classifier. Then, we compared a single run of WAF-A-MoLE against the best payload generated by the 100 unguided instances over time. Both the WAF-A-MoLE and the unguided fuzzers were configured to start from the payload `admin' OR 1=1#`, initially detected with 100% confidence by each classifier.

5.1 Assessment results

Figure 5 shows the evolution of the confidence score for each classifier. In each plot, we compare the best sample obtained by WAF-A-MoLE (solid line) and the best sample generated by all the 100 processes of the *unguided* fuzzer (dashed line).

The first group of plots (left) show the evolution of the confidence scores against the number of mutation rounds. The second group (right), shows the confidence score over the actual time of computation. In particular, we show the first 10 seconds of computation. Since some scores quickly degrade in the first milliseconds of computation, we report the x axis in log scale.

5.2 Interpretation of the results

Our experiments highlight a few facts that we discuss below.

Feature choice matters. As explained in Section 4.2, all the considered classifiers are based on syntactic features. However, different feature set change the robustness of a classifier. For instance, WAF-Brain quickly lost confidence when the payload mutated, because WAF-Brain is trained from uninterpreted, fixed-length sequences of characters and our mutation operators can enlarge a payload beyond the adequacy of the length assumed by WAF-Brain. Also Token-based classifiers do not perform well against mutations. The reason is that malicious and benign payloads overlap in the feature space. All SQLiGoT versions showed to be robust against the unguided approach. These classifiers use the SVM algorithm as some of the token based classifiers, but their feature set imposes more structure inside the feature representation. Hence, random mutations have a negligible probability to evade them. Instead, since WAF-A-MoLE relies on a guided strategy, it can effectively craft adversarial examples (although more effort is needed).

Finding adversarial examples is non-trivial. SQLiGoT classifiers resist the unguided evaluation as it is unlikely that a mutation can move the sample away from a plateau region where the confidence of being a SQL injection is high. The main reasons are: (*i*) SQLiGoT considered a large number of tokens (so reducing the collision problem that affects other classifiers, since the compression factor applied by the feature extractor is lower); (*ii*) The structure of the feature vector is inherently redundant, i.e., each pair of adjacent variables describe the same token; (*iii*) the models are regularized, hence the decision function is smoother between input points and it manages to generalize over new samples.

WAF-A-MoLE effectively evades WAFs. Moving randomly in the input space is not an effective strategy. WAF-A-MoLE finds adversarial samples by leveraging on hints given by classifier outputs. The guided approach accomplishes what the unguided approach failed to, by moving points away from plateaus and putting them in regions of low confidence of being recognized as SQL injection. Moreover, among the SQLiGoT classifiers, the *undirected unproportional* is the most resilient variant. Recalling the definition of the algorithm [25], the feature extractor assigns uniform weights to tokens in the same window instead of balancing the score w.r.t. the distance of the current token. Hence, the classifier gains some invariance over the sequence of extracted tokens, making it more robust to adversarial noise.

5.3 Discussion and limitations

Our experiments show that, starting from a target malicious payloads, WAF-A-MoLE effectively degrades the confidence scores of the considered classifiers. In this section we discuss implications and limitations of this result.

Generality of the experiments. As discussed in Section 4.1, the classifiers were trained with a dataset that we had to build from scratch. This has clear consequences on our experimental results. Hence, to extend the validity of our results, new experiments should be executed from other, real-world, datasets.

Another limitation is that we did not take into account the robustness of WAFs combining signatures *and* ML techniques, called *hybrid*. These systems are becoming more and more common.

Adversarial attacks mitigation. Demontis et al. [15] showed the effect of the presence of regularization when a classifier is under attack. Without regularization an attacker may craft an adversarial example against the target, due to the high irregularity of the victim function. Adding the regularization parameter has the effect of smoothing the decision boundary of the function between samples, reducing the amount of local minima and maxima. On top of that, the adversary needs to increase the amount of perturbations to craft adversarial examples. All models we trained have been properly regularized.

Grosse et al. [19] propose the so called *adversarial training*, that basically is a re-fit of the classifier also including the attack points. This defense system leads to better robustness against adversarial examples, at the cost of worse accuracy scores. Again, as shown by Carlini et al. [10] this is not a solution, but it may slow down the adversary in finding adversarial examples.

[11] https://www.digitalocean.com/

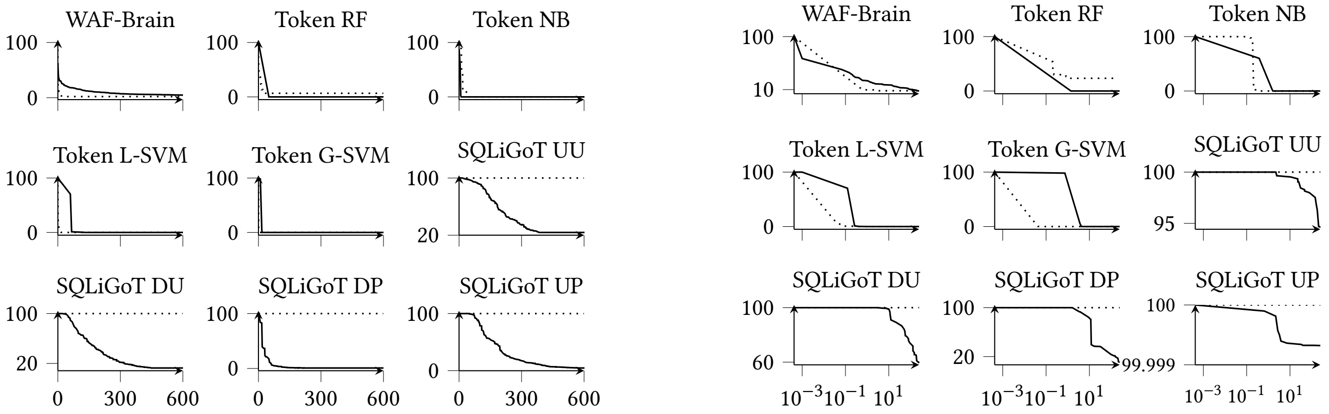

Figure 5: *Guided* (solid) vs. *unguided* (dotted) search strategies applied to initial payload `admin' OR 1=1#`.

6 RELATED WORK

In this section, we present some work related to WAFs, as well as evasion techniques that have been proposed to bypass them.

Attacks against signature-based WAFs: Appelt et al. [3, 4] propose a technique to bypass signature-based WAFs. Their technique is a search-based approach in which they create new payloads from existing blocked payloads. The problem with implementing a search-based approach in this context is hard: the obvious evaluation function for a payload against the target WAF is a decision function with values PASSED/BLOCKED. Search-based approaches perform poorly if the evaluation function has many plateaus. To mitigate this issue, the authors propose an approximate evaluation function which returns the probability of a payload of being "near" the PASSED or BLOCKED state. In the best case scenario, this function smooths the plateau and the search algorithm converges to the PASSED state.

Automata based WAFs: Halfond et al. [20] propose *AMNESIA*, a tool to detect and prevent SQL injection attacks. The algorithm works by creating a Non-Deterministic Finite Automa representing all the SQL queries that the application can generate. The main issues with this approach are that an attack can bypass it (*i*) if the model is too conservative and includes queries that cannot be generated by the application or (*ii*) if the attack has the same structure of a query generated by the application. Bandhakavi et al. [5] developed *CANDID*, a tool that detects SQL injection attempts via candidate selection. This approach consists of transforming queries into a canonical form and evaluating each incoming query against candidate ones generated by the application.

Machine learning WAFs: Ceccato et al. [11] propose a clustering method for detecting SQL injection attacks against a victim service. The algorithm learns from the queries that are processed inside the web application under analysis, using an unsupervised one-class learning approach, namely K-medoids [34]. New samples are compared to the closest medoid and flagged as malicious if their edit distance w.r.t. the chosen medoid is higher than the diameter of the cluster. Kar et al. [25] develop SQLiGoT, a support vector machine classifier (SVM) [13] that expresses queries as graphs of tokens, whose edges represent the adjacency of SQL-tokens. This is the classifier we used in our analysis. Pinzon et al. [33] explore two directions: visualization and detection, achieved by a multi-agent system called *idMAS-SQL*. To tackle the task of detecting SQL injection attacks, the authors set up two different classifiers, namely a Neural Network and an SVM. Makiou et al. [28] developed an hybrid approach that uses both machine learning techniques and pattern matching against a known dataset of attacks. The learning algorithm used for detecting injections is a Naive Bayes [29]. They look for 45 different tokens inside the input query, chosen by domain experts. Similarly, Joshi et al. [24] use a Naive Bayes classifier that, given a SQL query as input, extracts syntactic tokens using spaces as separator. The algorithm produces a feature vector that counts how many instances of a particular word occurs in the input query. The vocabulary of all the possible observable tokens is set a priori. Komiya et al. [27] propose a survey of different machine learning algorithms for SQL injection attack detection.

Evading machine learning classifiers: The techniques that are used in the state of the art are divided in two different categories: (*i*) gradient and (*ii*) black-box methods. For a comprehensive explanation of these techniques, Biggio et al.[8] expose the state of the art of adversarial machine learning in detail. The attacker can compute the gradient of the victim classifier w.r.t. the input they use to test the classifier. Biggio et al.[7] propose a technique for finding adversarial examples against both linear and non linear classifiers, by leveraging on the information given by the gradient of the target model. Similarly, Goodfellow et al.[18] present Fast Gradient Sign Method (FGSM), which is used to perturb images to shift the confidence of the real class towards another one. Papernot et al. [31] propose an attack that computes the best two features to perturb in order to most increase the confidence of it belonging to a certain class. This method leverages on gradient information too. If the attacker has information regarding a particular system, but they can not access it, they can try to learn a surrogate classifier, as proposed by Papernot et al.[30]. Many papers that craft attacks in other domains [14, 26, 35] belong to this category. If the attacker does not have access to the model, or they have no information on how to reconstruct it locally, they treat this case as a black-box optimization problem. Ilyas et al. [23] apply an evolution strategy to limit the number of queries that are sent to the victim model to craft an adversarial example. Xu et al. [37] propose a technique that uses a genetic algorithm for crafting adversarial examples that bypass PDF malware classifiers. Anderson et al. [2] evade different malware detectors by altering malware samples using semantics invariant transformations, by leveraging only on the score provided by the victim classifier.

7 CONCLUSION

We provided experimental evidence that machine learning based WAFs can be evaded. Our technique takes advantage of an adversarial approach to craft malicious payloads that are classified as benign. Moreover, we showed that WAF-A-MoLE efficiently converges to bypassing payloads. We show the results of this technique applied to existing WAFs, both via a guided and unguided approach. We leveraged on a set of syntactic mutations that do not alter the original semantics of the input query. Finally, we built a dataset of SQL queries and we released it publicly.

Our work highlights that machine learning based WAFs are exposed to a concrete risk of being bypassed. Future directions include testing WAFs based on other techniques such as hybrid ones, finding new mutations to improve our approach, and take advantage of our adversarial technique to improve detection of malicious payloads.

Acknowledgment. This paper was partially funded by EU H2020 research project SPARTA (grant agreement n. 830892).

REFERENCES

[1] Mark A Aizerman. 1964. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and remote control* 25 (1964), 821–837.

[2] Hyrum S Anderson, Anant Kharkar, Bobby Filar, and Phil Roth. 2017. Evading machine learning malware detection. *Black Hat* (2017).

[3] Dennis Appelt, Cu D Nguyen, and Lionel Briand. 2015. Behind an application firewall, are we safe from sql injection attacks?. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–10.

[4] Dennis Appelt, Cu D Nguyen, Annibale Panichella, and Lionel C Briand. 2018. A machine-learning-driven evolutionary approach for testing web application firewalls. *IEEE Transactions on Reliability* 67, 3 (2018), 733–757.

[5] Sruthi Bandhakavi, Prithvi Bisht, P Madhusudan, and VN Venkatakrishnan. 2007. CANDID: preventing sql injection attacks using dynamic candidate evaluations. In *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 12–24.

[6] Marco Barreno, Blaine Nelson, Russell Sears, Anthony D Joseph, and J Doug Tygar. 2006. Can machine learning be secure?. In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*. ACM, 16–25.

[7] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. 2013. Evasion attacks against machine learning at test time. In *Joint European conference on machine learning and knowledge discovery in databases*. Springer, 387–402.

[8] Battista Biggio and Fabio Roli. 2018. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition* 84 (2018), 317–331.

[9] Kay Henning Brodersen, Cheng Soon Ong, Klaas Enno Stephan, and Joachim M Buhmann. 2010. The balanced accuracy and its posterior distribution. In *2010 20th International Conference on Pattern Recognition*. IEEE, 3121–3124.

[10] Nicholas Carlini and David Wagner. 2017. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*. ACM, 3–14.

[11] Mariano Ceccato, Cu D Nguyen, Dennis Appelt, and Lionel C Briand. 2016. SOFIA: an automated security oracle for black-box testing of SQL-injection vulnerabilities. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 167–177.

[12] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[13] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine learning* 20, 3 (1995), 273–297.

[14] Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. 2019. Explaining Vulnerabilities of Deep Learning to Adversarial Malware Binaries. *arXiv preprint arXiv:1901.03583* (2019).

[15] Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. 2018. On the Intriguing Connections of Regularization, Input Gradients and Transferability of Evasion and Poisoning Attacks. *arXiv preprint arXiv:1809.02861* (2018).

[16] Jeremy D'Hoinne, Adam Hils, and Claudio Neiva. 2017. *Magic Quadrant for Web Application Firewalls*. Technical Report. Gartner, Inc.

[17] Parul Garg. [n.d.]. Fuzzing – Mutation vs. Generation. https://resources.infosecinstitute.com/fuzzing-mutation-vs-generation/. [Online; accessed 29-June-2019].

[18] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[19] Kathrin Grosse, Praveen Manoharan, Nicolas Papernot, Michael Backes, and Patrick McDaniel. 2017. On the (statistical) detection of adversarial examples. *arXiv preprint arXiv:1702.06280* (2017).

[20] William GJ Halfond and Alessandro Orso. 2005. AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 174–183.

[21] Thomas Hofmann, Bernhard Schölkopf, and Alexander J Smola. 2008. Kernel methods in machine learning. *The annals of statistics* (2008), 1171–1220.

[22] Ling Huang, Anthony D Joseph, Blaine Nelson, Benjamin IP Rubinstein, and JD Tygar. 2011. Adversarial machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*. ACM, 43–58.

[23] Andrew Ilyas, Logan Engstrom, Anish Athalye, and Jessy Lin. 2018. Black-box adversarial attacks with limited queries and information. *arXiv preprint arXiv:1804.08598* (2018).

[24] Anamika Joshi and V Geetha. 2014. SQL Injection detection using machine learning. In *2014 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*. IEEE, 1111–1115.

[25] Debabrata Kar, Suvasini Panigrahi, and Srikanth Sundararajan. 2016. SQLiGoT: Detecting SQL injection attacks using graph of tokens and SVM. *Computers & Security* 60 (2016), 206–225.

[26] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. 2018. Adversarial Malware Binaries: Evading Deep Learning for Malware Detection in Executables. *arXiv preprint arXiv:1803.04173* (2018).

[27] Ryohei Komiya, Incheon Paik, and Masayuki Hisada. 2011. Classification of malicious web code by machine learning. In *2011 3rd International Conference on Awareness Science and Technology (iCAST)*. IEEE, 406–411.

[28] Abdelhamid Makiou, Youcef Begriche, and Ahmed Serhrouchni. 2014. Improving Web Application Firewalls to detect advanced SQL injection attacks. In *2014 10th International Conference on Information Assurance and Security*. IEEE, 35–40.

[29] Melvin Earl Maron. 1961. Automatic indexing: an experimental inquiry. *Journal of the ACM (JACM)* 8, 3 (1961), 404–417.

[30] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. 2017. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 506–519.

[31] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. 2016. The limitations of deep learning in adversarial settings. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*. IEEE, 372–387.

[32] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[33] Cristian I Pinzon, Juan F De Paz, Alvaro Herrero, Emilio Corchado, Javier Bajo, and Juan M Corchado. 2013. idMAS-SQL: intrusion detection based on MAS to detect and block SQL injection through data mining. *Information Sciences* 231 (2013), 15–31.

[34] Leonard KAUFMAN Peter J RDUSSEEUN. 1987. Clustering by means of medoids. (1987).

[35] Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elovici. 2018. Generic black-box end-to-end attack against state of the art API call based malware classifiers. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 490–510.

[36] Andrey Nikolayevich Tikhonov. 1943. On the stability of inverse problems. In *Dokl. Akad. Nauk SSSR*, Vol. 39. 195–198.

[37] W Xu, Y Qi, and D Evans. 2016. Automatically Evading Classifiers: A Case Study on PDF Malware Classifiers. NDSS.

[38] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. Mutation-Based Fuzzing. In *The Fuzzing Book*. Saarland University. https://www.fuzzingbook.org/html/MutationFuzzer.html Retrieved 2019-05-21 19:57:59+02:00.