SOFIA: An Automated Security Oracle for Black-Box Testing of SQL-Injection Vulnerabilities

Mariano Ceccato
Fondazione Bruno Kessler
Trento, Italy
ceccato@fbk.eu

Cu D. Nguyen, Dennis Appelt, Lionel C. Briand
SnT Centre, University of Luxembourg
Luxembourg
{duy.nguyen,dennis.appelt,lionel.briand}@uni.lu

ABSTRACT

Security testing is a pivotal activity in engineering secure software. It consists of two phases: generating attack inputs to test the system, and assessing whether test executions expose any vulnerabilities. The latter phase is known as the security oracle problem.

In this work, we present SOFIA, a Security Oracle for SQL-Injection Vulnerabilities. SOFIA is programming-language and source-code independent, and can be used with various attack generation tools. Moreover, because it does not rely on known attacks for learning, SOFIA is meant to also detect types of SQLi attacks that might be unknown at learning time. The oracle challenge is recast as a one-class classification problem where we learn to characterise legitimate SQL statements to accurately distinguish them from SQLi attack statements.

We have carried out an experimental validation on six applications, among which two are large and widely-used. SOFIA was used to detect real SQLi vulnerabilities with inputs generated by three attack generation tools. The obtained results show that SOFIA is computationally fast and achieves a recall rate of 100% (i.e., missing no attacks) with a low false positive rate (0.6%).

CCS Concepts

•Security and privacy → Web application security; *Penetration testing;* **•Software and its engineering → Software testing and debugging;**

Keywords

Security testing; Security oracle; SQL-injection;

1. INTRODUCTION

SQL-injection (SQLi) vulnerabilities are amongst the top security threats to web-based software systems [8, 28]. Such vulnerabilities stem from defects in data validation procedures, such that when an attacker provides input values that contain fragments of SQL code, they eventually get injected into SQL queries that are executed on databases. With such a vulnerability, attackers can run arbitrary malicious code on databases to acquire or compromise sensitive data, such as medical records or financial transactions. The impact of SQLi exploitations can range from enabling fraud to compromising an organisation's reputation or even shutting down its activities. Though the root cause of SQLi has been well studied [10, 24], in reality, mostly due to time constraints and undisciplined development practices, many systems remain vulnerable to SQLi [12, 19, 28].

When engineering secure software systems and services, software testing is one of the main practices to detect faults as well as security vulnerabilities. Security testing (also called penetration testing) is a branch of software testing devoted to stress programs with respect to their security features, with the aim of identifying vulnerabilities. Security testing involves two major challenges, generating input values (referred to as *test payloads*), intended to exercise vulnerabilities, and evaluating whether such payloads manage to expose an actual vulnerability. The security oracle addresses the latter.

Security testing is highly expensive given the complexity of modern systems, typically providing a wide range of services, and the sophistication of attacks and exploitations. To reduce effort and cost, the research community has focused on automating security testing. Regarding SQLi, the test input generation problem has been extensively investigated and automated approaches are available [3, 10, 15, 16]. Automating the test oracle problem for SQLi vulnerabilities, however, remains an open problem. This is a significant obstacle to test automation, as manual oracles severely limit the number of test execution results a test team can process [5].

In this work, we present SOFIA, a Security Oracle for SQLi Attacks. Our goal is to satisfy three important requirements. First, it must be independent from known attack instances so that new types of attacks can be detected in the future. This is an important leap forward since existing solutions based on attack patterns can only detect publicly-known and documented attacks. Second, the oracles should not rely on knowledge about test input data or their generation algorithm in order to be usable with any given test generation tool. Third, our proposed oracle should not require the source code of the SUT, since we target black-box testing. This is often a mandatory requirement for external security testing (carried out by third-party penetration testers) or for systems whose source code is not available.

Most of the existing SQLi oracles either require known attacks in the learning phase [21] or access to source code [11] [6] [7] [14]. The few approaches that still meet all the three requirements [17] [26] are fundamentally different than our solution in ways that affect recall and false positive rates. Whereas they detect user inputs in SQL statements and compare them with user inputs observed at learning time, we prune data from SQL statements and compare their parse trees. By comparing structure instead of data, our goal is to enable SOFIA to yield high recall and low false positive rates. The main motivation is that modelling all possible safe data is highly difficult, if feasible at all, and false positives are caused by incomplete models. Further, we observed that a change in query structure is the most direct manifestation of an SQLi attack.

SOFIA is built using one-class machine classification. SQL statements issued by a SUT to its database are logged and parsed to create SQL parse trees, which are fed to a clustering algorithm. The *Tree edit distance* is used to measure the distance amongst parse trees. Our approach consists of two phases: *Training* and *Testing*. During training, *legitimate* SQL statements, which are obtained from regular executions, are grouped into clusters of similar statements. We refer to this set of clusters as a *safe model*. Such a model represents legitimate database accesses in the absence of attacks.

In the testing phase, when test inputs trigger new SQL statements, our oracle assesses whether the statements can be assigned to the clusters of the safe model. In the positive case, we can assert that the statements are safe and no vulnerability is reported. Otherwise, such statements are classified as anomalous, and hence, vulnerability alerts are reported. We have carried out an experimental evaluation in terms of false positive rate, recall rate, and computational cost on six real applications and with three different attack generation tools. The obtained results show that the proposed oracle achieves a very low false positive rate (0.6%) and misses no attack (100% recall) with a low computational overhead.

The proposed oracle is meant to support security testing, by classifying SQL statements triggered by test cases as legitimate statements or as SQLi attacks. However, since it relies on a black-box strategy and is trained only on legitimate executions, it could be also deployed as a database firewall in production to filter SQL statements and block SQLi attacks before they are actually executed. Investigating such potential application is out of the scope of this paper though and we present and assess the proposed oracle only in the context of security testing.

The next section provides background and discusses related work. In Section 3, we discuss the requirements and our strategy for the security oracle. In Section 4 we present in detail our approach. Section 5 reports our experiments to assess the accuracy and speed of the oracle. Finally, Section 7 concludes our work.

2. BACKGROUND

SQL Injection Vulnerabilities. In systems that use databases, the SQL statements that query the back-end database are usually treated by the native application code as strings. These strings are formed by concatenating different string fragments based on user choices or the application's control flow. For example, an SQL statement can be formed as follows:

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

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

SQLi is an attack technique in which attackers inject malicious SQL code fragments into input parameters that lack proper validation or sanitisation. An attacker might construct input values in a way that changes the behaviour of the resulting SQL statement and performs arbitrary actions on the database (e.g. exposure of sensitive data, insertion or alteration of data without authorisation, loss of data, or even taking control of the database server).

In the previous example, if the input *$country* received the attack payload *' or 1=1* --, the resulting SQL statement is:

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

The clause *or 1=1* is a tautology, i.e., the condition will always be true, and is thus able to bypass the original condition in the *where* clause, making the SQL query return all rows in the table. Hence, the above piece of code is vulnerable to SQLi attacks.

In security testing for SQLi, one important task is to generate such attack payloads so that they can circumvent projection layers (e.g., web application firewalls or input filtering), inject into vulnerable SQL code, and trigger executable SQL statements [15, 2]. Another equally important task is the oracle problem, that is how to assess whether a test payload detects an SQLi vulnerability. The traditional way, in a black-box testing context, is to analyse the responses of the SUT to which legitimate or attack inputs are sent or to analyse the triggered SQL statements for common attack patterns. We elaborate the limitations of such techniques further in the next section.

Tree Edit Distance. Ordered labelled trees refer to a tree structure in which nodes are labelled and edges capture predecessor-successor relationships amongst nodes. The left-to-right order amongst siblings is also significant to the semantics of the trees. Parse trees that structure sentences or programs according to some context-free grammars are ordered labelled trees.

Transforming one ordered labelled tree (or just tree for brevity) into another involves three basic types of edit operations: *changing a node label*, *delete a node*, and *insert a node*. As an example, taking the trees t_1 and t_2 in Figure 1, transforming t_1 into t_2 can be performed with a sequence of four edit operations: *change e to k, delete c, delete d, add h*, or alternatively: *change e to k, change d to h, delete c*.

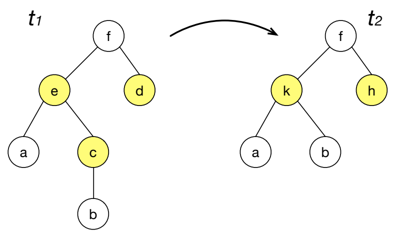


Figure 1: An example of two ordered labelled trees.

Formally, each edit operation o_i is assigned a cost (usually one unit). The cost of a sequence of edit operations $S_j = \langle o_1, o_2, \ldots, o_N \rangle$ is the sum of the cost of all operations o_i. Since there are usually alternative sequences to transform a tree into another, the tree edit distance between two trees is the minimum cost amongst possible sequences. When the cost of all edit operations is equal to one, then the edit distance between two trees is the number of operations of the shortest sequence that transforms one tree into the other.

3. REQUIREMENTS AND GENERAL STRATEGY

3.1 Security Oracle Requirements

Most of the classifiers used as security oracles are learned on a training set that contains both positive and negative examples (attacks and legitimate executions) [4]. However, these approaches are expected to suffer from two main limitations, namely, (i) the availability of attacks, and (ii) the representativeness of attacks, especially when the sample is small as in most practical contexts.

Documented attacks are usually unavailable for many systems. Some ethical attackers make their techniques and payloads available on the Internet, but software developers are usually unaware of them. Even if they are, the number of known attacks is limited. We also need to account for unknown, new attacks that may appear after the system's deployment. As a result, a security oracle would be much more beneficial for testing (and also in other contexts like monitoring in production) if it does not require the availability of documented attacks.

In addition, even when available, attacks used to train security oracle classifiers are often expected to be *representative* of possible attacks that could target a system. Unfortunately, this is normally not the case as new attacks are being introduced at a very fast pace. Therefore, a classifier should not rely on documented attacks at the risk of being ineffective with new ones. Thus, the first requirement for a security oracle is the following:

Requirement Req_1*: The security oracle should be independent from known instances of successful attacks.*

Often in security testing, oracles depend on knowledge about what attack generation algorithm and what test input data have been used to determine the expected output if there is a vulnerability. For example, some oracles classify a test as a successful attack when the execution output contains the same attack payloads as the inputs [15]. This strategy suffers from the observability problem as the output can be masked by a generic error message, or worse, the impact of the successful attack cannot be easily observed by the tester, e.g., like in *second order* SQLi attacks. Furthermore, this strategy, because it is specific to test generation algorithms or input data, limits the portability of the oracle. As a result, it may not work with most attack generation tools, without additional adaptation overhead. It is desirable to define an oracle that is independent from attack generation strategies so that it can be used with many attack tools and a variety of attack generation approaches. The second requirement is, thus:

Requirement Req_2*: The security oracle has no knowledge on what input data are used to test the system.*

Often, systems are written using frameworks and third part libraries. Since commercial libraries are rarely distributed with source code, a white-box approach in the generation of a security oracle is of limited applicability in real industrial settings. In many contexts, such as for third-party penetration testers, access to source code is not an option. Thus, the third requirement is:

Requirement Req_3*: The security oracle should not rely on the source code of the SUT.*

3.2 Our Strategy

Our strategy to create a security oracle for SQLi vulnerabilities that satisfies the above requirements relies on a black-box, security safe model, that is a model of safe execution inputs.

Safe model: To be independent from known attacks and attack tools, we decided to exclude attacks from the training set used to learn the security oracle. We propose a security oracle that only takes into consideration legitimate executions and builds a model of safe SQL statements. Tests will be classified as legitimate if they generate SQL statements satisfying this safe model and potential attacks otherwise.

Black-box: The SQL statements sent by the SUT to the back-end database are the only features considered by the oracle, either in the training phase and in the testing phase. As a result, such an oracle can be deployed to test systems developed in many languages and has no dependency or limitation regarding specific development frameworks.

The proposed safe model depends on the specific legitimate executions that are being considered. However, using classical black-box testing techniques, it is much easier to generate a large number of representative legitimate inputs than it is to generate attacks. The next section explains in detail the process to construct our security oracle.

4. SOFIA: THE SECURITY ORACLE

The procedure to build and apply the security oracle is summarized in Figure 2. It consists of two phases, *Training* and *Testing* including five steps: *Parsing*, *Pruning*, *Computing Distance*, *Clustering*, and *Classification*. These two phases share the first three steps. *Clustering* is exclusively part of training while *Classification* is exclusively part of testing.

The process starts with a set of SQL statements, obtained from safe executions of a SUT, either by executing functional tests or by monitoring regular system executions. The SQL statements are parsed and the parse trees represent the objects to be classified. The fact that our oracle uses only legitimate statements allows us to avoid the task of manual labelling training data (as legitimate statements or attacks), as often required by other supervised techniques.

Information from the parse trees, which is specific to concrete SQL statements and irrelevant for detecting attacks, is removed by pruning the parse trees. This helps not only in reducing the number of unique trees to be clustered and better scale, but also improves the overall attack detection performance.

Clustering relies on the edit distance amongst pruned parse trees. *Clustering* is used to group together similar SQL statements. Statements with low distance are assigned to the same cluster, while statements with larger distance are assigned to different clusters. The final safe model consists of the optimal partition of SQL statements computed by clustering. Note that the training process that creates safe models from SQL execution logs takes place only once. Safe models are then ready to support security testing in detecting SQLi vulnerabilities.

New statements triggered by executing security tests will be classified using the safe model, by assessing their distance to the centres of the clusters. If a new statement is close enough to a cluster centre, it satisfies the model and is classified as benign statement. Conversely, in the case a new SQL statement does not fit into any existing cluster, it is considered anomalous and classified as a potential attack. Further details of this process are provided in the sections that follow with the help of a running example.

Regarding the defined requirements for the security oracle, Requirement Req_1 (independence from known attacks) is satisfied since our approach relies only on logs of benign executions. Moreover, the fact that only database logs are considered by the oracle ensures that requirement Req_3 is also achieved: no access to the application source code is required. Our classification procedure complies with requirement Req_2 since it exclusively relies on the SQL statements sent to the database, and not the test case input values.

4.1 Training Data

Training data are used to construct the security model. However, differently from other approaches that require both attack and

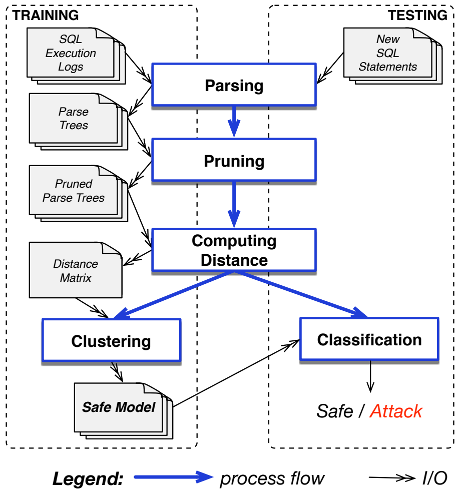


Figure 2: The overview SOFIA: the training process for learning safe models from SQL execution logs; the classification process for classifying new SQL statements.

legitimate SQL statements, our approach only relies on the latter to learn a classifier. In fact, our goal is to construct a model of *safe* executions, and classify as *anomalous* everything that does not conform to this model.

Training data are collected by executing the functional test suite of a SUT, or by monitoring its usage during production or acceptance testing. We collect execution logs containing SQL statements executed on the database. To this end, different technologies can be used depending on the underlying DBMS. For *mysql* we use the *mysql-proxy* tool[1] that monitors all traffic to and from *mysql* databases. This solution can be ported to other DBMSs as well. For example, for Java applications that use a JDBC driver to connect to Oracle Database, we can customise the driver to log SQL statements.

Let us use a running example where the execution log includes the SQL statements shown in Figure 3. While the three statements query the *user* and *password* columns from the table *users*, they differ from one another in the where conditions.

```
stmt1: select user, password from users
                         where id = 1;

stmt2: select user, password from users
                         where id = 2;

stmt3: select user, password from users
               where id = 4 and role = 1;
```

Figure 3: Three samples of SQL log of the running example.

4.2 Parsing

An SQLi attack modifies the semantic of SQL statements, usually by replacing a value with a piece of SQL code, for example by adding a tautology to the *where* clause, or by injecting an additional *select* or *union* statement. These injections, if successful, result in SQL queries that are valid SQL statements according to the SQL grammar, but yet have different parse trees. We resort to the parse trees of SQL statements to detect SQLi attacks.

In our work, we rely on the General SQL Parser[2] (GSP for short). GSP is a Java library that supports various DBMSs, including Oracle, SQL Server, DB2, MySQL, Teradata, and Access. Output parse trees are stored as XML documents for other analysis.

Figure 4 shows the parse trees for the SQL statements of our running example. As we can see in the figure, the parse trees contain information that is irrelevant and therefore detrimental to the process of learning a classifer through clustering, e.g., specific user ids. Indeed, some elements in the trees are very specific to the captured SQL executions and are irrelevant for the detection of attacks. The pruning process, described next, aims at removing such irrelevant information from the parse trees.

4.3 Pruning

Pruning could be done according to different strategies depending on what piece of information should be removed. To decide about the most effective pruning in our context, we should consider how attacks are typically carried out. SQLi attacks aim at altering SQL statements by replacing data with a new piece of SQL code, i.e. a string or numeric literal is replaced by code. Thus, two legitimate SQL statements collected during the execution of the same feature but with different input data should only differ in terms of data values. Instead, a legitimate statement and an attack statement should differ not only with respect to data but also the SQL command structure in the maliciously injected part.

Based on the above consideration, we decided to prune data values in parse trees: We replace all the constant numeric and string values in the tree with the same placeholder (e.g. with the empty string or with the constant zero). As a result, statements that just differ in values are characterised by a single pruned tree.

Figure 5 shows the pruned parse trees of our running example. Nodes with data values are replaced with a place holder (the `?` character). Since *stmt1* and *stmt2* only differ in data value of the *id* attribute, their pruned versions are equivalent. For learning purposes, redundant versions of the same pruned trees will be discarded. Out of the three statements of the training set for the running example, only two distinct pruned trees will be considered to construct the safe model.

Our pruning procedure is straightforward. We analyse the XML parse trees and replace the content of the leaf nodes of type *Tconstant*, which contain concrete data values, with a placeholder.

4.4 Computing Distance

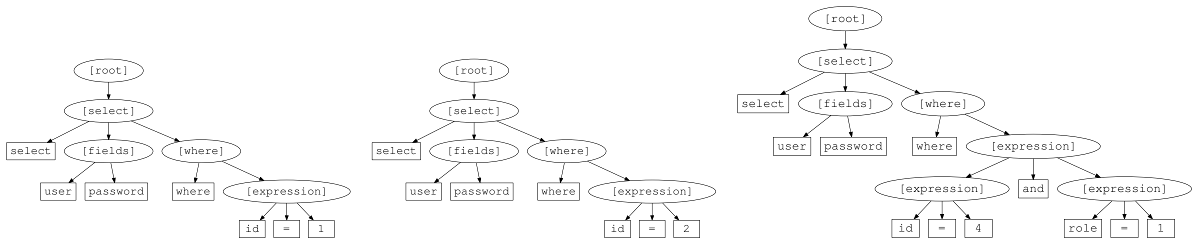
Parse trees of SQL statements are ordered and labelled trees. A metric of tree edit distance for this class of trees has been proposed by Zhang et al. [23], as discussed in Section 2. Zhang et al. have also proposed a fast algorithm to calculate tree edit distances in a polynomial time complexity. Our work makes use of a tool called *approxlib*[3], that implements this specific algorithm.

Let us consider the SQL statement of the attack in Figure 6(a), for which the corresponding (pruned) parse tree is shown in Figure 6(c). We note that this tree is quite similar to the parse tree of the legitimate statement 3 in Figure 5(b). In fact, we note that these trees are more similar (distance is 11) than the two legitimate statements *stmt1* and *stmt3* (distance is 22). This example highlights the fact that parse tree distance alone is not enough to detect attacks. We need to infer a classification amongst legitimate statements, in this case using clustering to group similar trees, and compare a candidate attack with its closest cluster. The underlying rationale

[1]https://dev.mysql.com/downloads/mysql-proxy

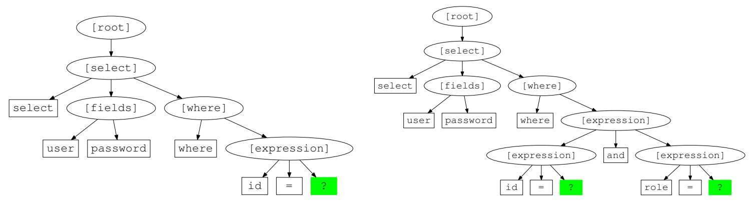
[2]http://sqlparser.com/

[3]http://www.cosy.sbg.ac.at/ augsten/src



(a) stmt1 (b) stmt2 (c) stmt3

Figure 4: Parse trees of the SQL statements.



(a) stmt1 and stmt2 (b) stmt3

Figure 5: Pruned parse trees of the example SQL statements.

is that, though a legitimate SQL statement does not obviously need to be similar to all legitimate statements resulting from test executions, there should be a cluster with similar statements. Once parse tree distances are computed amongst all the pairs of pruned parse trees, an algorithm can be applied to generate an optimal set of clusters, as described next.

4.5 Clustering

We address our clustering problem using the k-medoids algorithm [22]. This algorithm is a variant of k-means clustering, based on the search for k representative samples, namely the *medoids*, amongst the observations of the dataset. After finding a set of k medoids, k clusters are constructed by assigning each observation to the nearest medoid. We need to find k representative samples that minimize the sum of the dissimilarities of the observations to their closest representative object. Furthermore, we consider a measure of cluster *diameter*. The diameter of a cluster is the maximal distance between the observations in the cluster and its medoid. To build a safe model, training data are clustered into k clusters, characterised by their medoids and diameters.

The adoption of k-medoids clustering is more appropriate in our context than the standard k-means [13] approach. K-means involves the notion of *mean* point, which in our case would mean an average tree for all the observations in the same cluster. Since such a hypothetical tree is insensible in our context, we adopt k-medoid, instead. It picks a representative element for each cluster (i.e., the *medoid*), instead of computing a fictitious average tree.

It is very important to identify the appropriate number of clusters k. Clusters should be small enough to distinguish attacks from legitimate statements based on parse tree distances, but clusters should also be large enough to capture representative groups of similar legitimate statements. More specifically, a too small value of k would elaborate a partition that contains few large clusters. A large cluster would contain very different parse trees, with large distances from each other, and its medoid would not be representative of all the members of the clusters. Also, with large clusters, the distance between an attack and the cluster medoid may be comparable to the cluster diameter. Thus, actual attacks would be wrongly classified as legitimate statements (false negatives).

On the other hand, a large number of clusters k may result in many clusters that contain too few elements to be representative and enable reliable comparisons with new parse trees. False alarms (false positives) may result from new legitimate statements whose tree is at a distance from the closest medoid that is higher than the cluster's diameter.

To decide the most appropriate number of clusters (i.e., the value of k), we adopt a standard approach, the Akaike information criterion (AIC) [1, 18]. It entails balancing the trade-off between the goodness of fit of the model and the size of the model. The underlying rationale is to increase the complexity of the model (i.e., k increases) as far as the gain in precision is high, and stop when the increase in precision is not significant. To achieve this objective, we adopt a penalty factor for each new cluster. To determine the number of clusters in this way, we select the best k that minimizes the fitness function f composed of two terms: (i) *distortion*, a measure of the extent to which SQL statements deviate from the prototype of their clusters (e.g., *RSS* for k-medoids); and (ii) the *model complexity* that is proportional to the number of clusters:

$$f(k) = RSS(k) + 2 \cdot M \cdot k$$

Where RSS is the residue sum of square, i.e. the error that we commit by approximating each observation in a cluster by the corresponding medoid, and M is the dimensionality of the vector space. In our case M=1, because the only feature used in clustering is the tree-edit distance.

The resulting optimal set of clusters, each associated with a medoid and diameter, represents the safe model used as an oracle to classify newly executed SQL statements. Table 1 shows the final clustering configuration for the running example.

Table 1: Clustering results for the running example.

Cluster	Medoid	Elements
1	*stmt1*	*stmt1*, *stmt2*
2	*stmt3*	*stmt3*

4.6 Classification

Intercepted SQL statements undergo the testing (classifying) process depicted in Figure 2: they are parsed, pruned, and eventually classified as safe or malicious. The classification procedure is described with respect to the test sample (a malicious statement) shown in Figure 6 and consists of the following steps:

1. Parsing and Pruning: When the test SQL statement (Figure 6(a)) is intercepted by our database proxy, it is parsed (Figure 6(b), malicious injected code highlighted in red) and pruned (Figure 6(c), pruning highlighted in green) as described previously.

2. Computing Distance: The tree edit distance is used to identify the closest cluster in the safe model. To compute this result, the pruned parse tree T of the test (Figure 6(c)) is compared to the pruned parse tree of each medoid. In our example, the distances from the medoids are shown in Figure 6(d) as 18 and 11, respectively.

The medoid with the smallest distance from the test (*stmt3* in our example) determines the nearest cluster (cluster 2). Determining the nearest cluster is fast since the medoids' parse trees are pre-computed and only k tree edit distances need to be evaluated. In the example only two comparisons are required.

3. Distance-versus-diameter classification: We check whether the test fits the nearest cluster by comparing its diameter and the distance between the test parse tree and medoid. Alternative measures could be used, such as the distance corresponding to the 95th percentile of cluster elements instead of its diameter, to deal with potential outliers. However, this is out of scope for this work.

In our example, the diameter[4] of cluster 2 is equal to 0. We compare the distance between the test T and the nearest medoid (distance is 11) with the diameter (equals to 0). When the distance to the medoid is smaller than or equal to the diameter, the test is deemed to fit this cluster and it is classified as a safe execution. However, in our example the test falls outside of the cluster border (distance $>$ diameter) and is classified as a potential attack.

5. EXPERIMENTAL EVALUATION

This section presents the experimental evaluation designed and conducted to assess the proposed security oracle.

5.1 Research Questions and Variable Selection

We investigate the following main research questions:

- **RQ$_1$**: *How accurate is SOFIA in classifying legitimate SQL statements and SQL-injection attacks?*
- **RQ$_2$**: *How fast is SOFIA in classifying SQL statements as legitimate or attacks?*
- **RQ$_3$**: *How does SOFIA compare to main-stream alternative tools in terms of accuracy and speed?*

The first research question concerns the accuracy of the oracle in classifying SQL statements. Missed attacks may lead to unaddressed security defects and false alarms lead to significant wasted effort, which should remain within reasonable bounds.

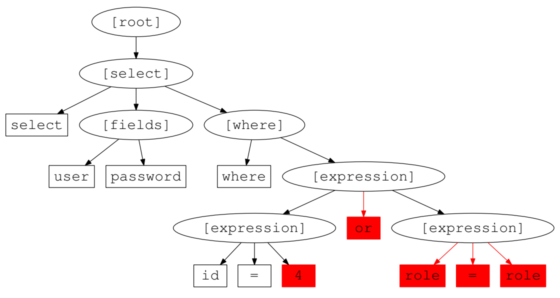
The second research question is about the amount of time taken by the classifier to make a decision on a newly observed SQL statement. Fast run-time classification of attacks during testing is important to support efficient test automation.

The third research questions is meant to compare SOFIA with available and comparable alternative solutions, which can be considered a baseline on which to improve, both in terms of classification accuracy and speed.

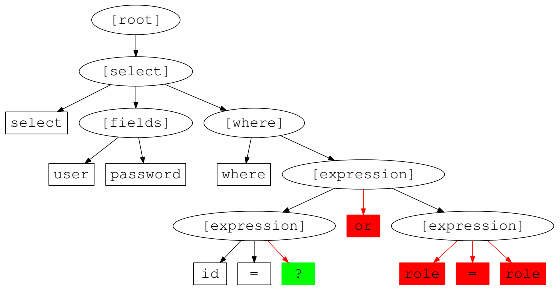
[4]Because of the small size of the running example, each cluster contains just one pruned parse tree, so diameter is equal to 0.

select user, password **from** users
where id = 4 **or** role = role;

(a) Statement



(b) Parse tree



(c) Pruned parse tree

Cluster	Medoid	Diameter	Test distance
1	stmt1	0	18
2	stmt3	0	11

(d) Cluster diameters and distance

Figure 6: Classification of a malicious statement.

Accuracy in our context is characterised by attack detection rate (we want to detect as many attacks as possible) and false positive rate. We need to minimise false positives as they trigger unnecessary manual analysis, which is expensive. The more false positives, the more analysis effort is being wasted and the scalability of the approach is being compromised. Accuracy is quantified by the standard *Recall* and *FPR* (False Positive Rate) metrics from information retrieval:

- *True Positives (TP):* The number of actual attacks that are correctly classified by the oracle as attacks;
- *False Positives (FP):* The number of legitimate statements that are incorrectly classified by the oracle as attacks;
- *True Negatives (TN):* The number of legitimate statements that are correctly classified by the oracle as safe;
- *False Negatives (FN):* The number of actual attacks that are incorrectly classified by the oracle as safe statements;
- **Recall:** The ratio between the correctly detected attacks and all the actual attacks $\frac{TP}{TP+FN}$;
- **FPR:** The ratio between false positives and all the actual legitimate statements $\frac{FP}{FP+TN}$;

Classification time is measured as follows:

- **C-Time:** Amount of time spent by the classifier to make a decision on a test outcome.

Time is measured by instrumenting the oracle. System time is probed before starting and after concluding the complete classification procedure for each SQL statement. It includes the amount of time spent for parsing the statement, pruning its tree, and the amount of time for computing its distances to the medoids of the safe model oracle.

The amount of time required to train the oracle is less interesting because training is done just once in a while and has therefore limited practical implications. It will not be further discussed here.

5.2 Subject Applications

The subject applications considered in this study are web applications and web services that use an SQL relational database. Moreover, since the security oracle will be assessed on real vulnerabilities, we selected applications by inspecting their bug-tracking systems and the *Common Vulnerabilities and Exposures* repository[5] that keeps track of publicly known vulnerabilities and exposures. The chosen subject applications contain real SQL-injection vulnerabilities.

The applications considered in the study are:

- *HotelRS*: written in PHP, *HotelRS* is a service-oriented based system providing web services for room reservation. It was developed and used by Coffey et al. [9].
- *SugarCRM*: written in PHP, *SugarCRM* is a popular customer relationship management system[6].
- *Taskfreak*: written in PHP, *Taskfreak* is a web project management application[7].
- *Theorganizer*: a web application that supports management and organisation of the activities in a personal agenda[8]. The server is written in Java (using Servlets, J2EE and Spring JDBC).
- *Wordpress*: written in PHP, *Wordpress* is a popular blogging and news publishing platform[9]. *Wordpress* has many utility plugins that are vulnerable to SQLi. We have two variants of *Wordpress*, one with the *newstatpress* plugin[10] that provides access statistics to *Wordpress*; the other with *landing-pages*[11], a plugin for customising templates and attracting more visits to blogging sites.

We use the test suites of the subject applications for generating training data. *Wordpress* comes with a large test suite of more than 3700 phpunit test cases. For *Taskfreak* and *Theorganizer*, we reuse the test suites generated by available techniques [25, 20] and its accompanying tool[12]. For the remaining two, *HotelRS* and *SugarCRM*, we manually defined test suites that exercise all operations of their web services with various domain inputs.

Our reliance on real vulnerabilities in applications makes our results more representative of the current situation though it is impossible to predict what these results will be with future types of vulnerabilities. However, as described above, because our approach does not learn from specific attacks and relies on learning to characterise safe statements, we hope that the safe model will be able to handle future types of vulnerabilities as well.

5.3 Attack Generation

To evaluate the classifier, both legitimate executions and attacks are required. However, our oracle is independent of the input data generation strategy adopted to generate the attacks. Thus, we will evaluate the accuracy of the oracle with diverse attack generation tools:

- *burpsuite*: A commercial security testing tool suite[13]. It has a vulnerability scanner that targets many types of vulnerabilities, including those in the OWASP top 10 [28]. For detecting SQLi, *burpsuite* (version 1.6.23) has a fixed list of 134 built-in SQLi test payloads, such as:

  ```
  'a or 1=1-- | //* | replace | drop table
  ```

  When scanning for SQLi vulnerabilities, *burpsuite* uses these payloads as request parameters and submits them to a target system. It then analyses the obtained responses from the system to detect SQL code or error messages in order to report SQLi issues.
- *sqlmap*: A popular open source tool for penetration testers to detect and exploit SQLi vulnerabilities[14]. It supports various database management systems and implements many heuristics to generate test payloads for different types of SQLi, including *boolean-based blind, time-based blind, error-based, UNION query-based, stacked queries and out-of-band*.
- *Xavier*: A framework for the automated testing of web services for SQLi vulnerabilities [3] [2]. Powered by a grammar developed specifically for SQLi attacks and machine learning, *Xavier* can generate diverse test payloads that can bypass web application firewalls and detect SQLi vulnerabilities.

5.4 Alternative Oracles

Apart from assessing SOFIA on diverse applications, it is interesting to investigate how it fares when compared to existing tools with similar goals. We found only two alternative tools: *antiSQLi* and *GreenSQL*[15].

antiSQLi is a tool provided by the vendor of the SQL parser we use in our work, which takes log files containing SQL statements as inputs, and reports whether their content is classified as attacks or legitimate statements.

GreenSQL is a popular database security solution for controlling database accesses, blocking SQLi attacks, among other features. It intercepts communications between applications and databases, learns patterns of regular SQL statements, and then, blocks malicious ones from getting to databases under protection.

5.5 Experimental Procedure

To collect SQL statements, we install and configure the subject applications, each on a separate virtual machine having *mysql* and *mysql-proxy* ready. *mysql-proxy* helps intercept and log all the SQL statements that an application sends to its database. We, then, execute the test suite that comes with each application to collect legitimate executions, i.e., safe statements. After that, we run the attack tools to generate attacks. The logs of attack tools are manually analysed and statements are labelled as attack or safe. Such analysis is required as safe statements can result from attacks since the system might perform routine updates or run additional queries before executing the attack statements. Note that the labelling task is only relevant for our assessment purposes; it is not needed in the real usage of the oracle.

We adopt a 10-fold validation strategy to check our oracle on different partitions of training and testing data. The training data of each subject application, consisting exclusively of safe statements,

[5]https://cve.mitre.org
[6]http://www.sugarcrm.com
[7]http://www.taskfreak.com
[8]http://www.apress.com/9781590596951
[9]https://wordpress.org
[10]https://wordpress.org/plugins/newstatpress
[11]https://wordpress.org/plugins/landing-pages
[12]http://selab.fbk.eu/magic
[13]http://portswigger.net/burp
[14]http://sqlmap.org
[15]http://www.greensql.com

is divided randomly into 10 sets of approximately the same size, to form 10 partitions:

- Nine sets of legitimate statements represent the *training set*. They are used to train the safe model of the oracle. In our case, this is done only on legitimate statements and there is no need to split attacks across training and testing sets;
- The remaining set of legitimate statements is merged with the attacks to form the *testing set*. The oracle is used to classify each entry in the testing set using the safe model.

This process is iterated 10 times, once per each of the 10 possible partitions. The classification elaborated by the oracle for the testing set is compared with the actual labelling, to evaluate the oracle accuracy. The 10-fold validation, including training and testing for all subject applications, was executed using a HPC (high-performance computing) system at the University of Luxembourg [27], where the CPU speed on nodes is 2.26GHz, and a 4Gb RAM was available to each process.

5.6 Experimental Results

Table 2 reports the number of SQL statements per application that have been considered in our study. The first two columns contain the name of the application and the tool used to generate the attacks, respectively. The subsequent columns report the number of legitimate statements and the number of successful attacks. The last two columns indicate the number of distinct pruned trees for legitimate statements and attacks. We cannot explore all combinations subject applications and tools because of certain application characteristics (web-based and web services) and the intended usage of the tools: *Xavier* targets web services while the others target standard web-based applications. In total, we obtain nine datasets for the experimental evaluation.

Table 2: Summary of the datasets used in our experiment: nine datasets obtained from six applications and three attack tools.

Application	T. Tool	#Legit.	#Attack	#Pruned Legit.	#Pruned Attack
HotelRS	Xavier	10,392	1,871	2,124	442
SugarCRM	Xavier	100,683	196	52	78
Taskfreak	burpsuite	7,502	3	29	2
Taskfreak	sqlmap	7,503	4	30	2
Theorganizer	burpsuite	1,516	28	27	17
Theorganizer	sqlmap	1,616	27	25	18
Wordpress-newstatpress	burpsuite	196,556	314	860	277
Wordpress-newstatpress	sqlmap	148,325	4	809	2
Wordpress-landingpage	sqlmap	171,487	170	843	65

We can observe, based on the data shown in Table 2, that the datasets used in our experiments are diverse in terms of the number of safe statements, ranging from 1k to 170k. Likewise, the number of attack statements generated by different tools varies from three to 1,871 attacks. Also, we can see that the number of parse trees after pruning is significantly reduced. For example, for subject *Wordpress-landingpage* and tool *sqlmap*, there were originally more than 171k safe statements and 170 attacks. However, after pruning, there are only 843 safe and 65 attack cases left, respectively. Overall, the percentage of reduction for all subject applications after tree pruning ranges from 79% to more than 99%. As a result, pruning helps reducing the time required by SOFIA, especially for training. Note that *burpsuite* cannot generate any attack on *Wordpress-landingpage* and, therefore, this pair is not investigated.

Table 3 provides experimental results. For each application, the performance of SOFIA is measured using Recall, false positive rate, and C-Time, as previously described. The values in the table represent the average over the 10 partitions of training/testing data and executions.

Table 3: Results of our approach: data averaged from 10-fold cross validation. *T.(ms)* is classification C-Time measured in millisecond.

App./*tool*	TP	FP	TN	FN	Recall	FPR	T.(ms)
HotelRS/ *Xavier*	1,871	0.0	1,039.2	0	**1.0**	**0.000**	25.14
SugarCRM/ *Xavier*	196	1.0	10,067.3	0	**1.0**	**0.000**	463.77
Taskfreak/ *burpsuite*	3	0.3	749.9	0	**1.0**	**0.000**	48.13
Taskfreak/ *sqlmap*	4	0.4	749.9	0	**1.0**	**0.001**	152.91
Theorganizer/ *burpsuite*	28	0.9	150.7	0	**1.0**	**0.006**	26.11
Theorganizer/ *sqlmap*	27	0.5	161.1	0	**1.0**	**0.003**	29.07
Wordpress-newstat/ *burpsuite*	4	28.5	14,804.0	0	**1.0**	**0.002**	33.70
Wordpress-newstat/ *sqlmap*	170	29.3	17,119.4	0	**1.0**	**0.002**	28.92
Wordpress-landingpage/ *sqlmap*	314	28.0	19,627.6	0	**1.0**	**0.001**	20.30

Regarding **RQ**$_1$, results in Table 3 show that SOFIA yields a perfect Recall of 100% for all the subject applications and achieves a very low false positive rate across all applications (0.006 at the highest). When we consider the absolute number of false positives (FP), they are mostly below 1.0 on average. The highest FP is only 29.3, therefore suggesting that manual analysis is feasible even in the worst case.

Thus we can provide a clear answer to **RQ**$_1$:

SOFIA yields a very high accuracy when classifying both legitimate SQL statements and attacks.

Moreover, considering **RQ**$_2$, the time required to classify a new statement is small, with an average across case studies of 92ms and a median of 29ms. For most of the case studies, classification takes around 30ms per statement, with the exception of Sugar-xavier that takes more than 400ms. The reason for this difference is that, on average, the parse trees of SQL statements used by Sugar are larger and thus lead to longer tree-edit distance calculations.

Thus we can answer **RQ**$_2$:

SOFIA is fast in classifying SQL statements, taking on average 92ms per classification.

Furthermore, to answer **RQ**$_3$, we compared SOFIA to *antiSQLi* and *GreenSQL.*

Following the same procedure as for SOFIA, we ran these two industrial tools against all nine datasets and measured their accuracy and time. *GreenSQL* was given the same training data that had been used to train SOFIA. *antiSQLi* inspects SQL statements based on its own SQLi filters and, therefore, no training was needed. The same testing data were then checked by *GreenSQL* and *antiSQLi*.

For each dataset, TP and FN were measured by counting the number of attack SQL statements correctly classified as attacks or incorrectly classified as safe, respectively. Likewise, we measured the average number of safe statements classified as attacks (FP) and safe (TN). Further, we measured the average execution time the tools required to parse and check a statement.

Table 4: Results of *antiSQLi* and *GreenSQL*. *T.(ms)* is the average time in millisecond the tools need to process one statement.

App/*tool*	TP	FP	TN	FN	Recall	FPR	T.(ms)
antiSQLi							
HotelRS/ *Xavier*	1,579	827.0	212.2	292	**0.84**	**0.796**	320
SugarCRM/ *Xavier*	50	1,237.6	8,830.7	146	**0.25**	**0.123**	314
Taskfreak/ *burpsuite*	1	12.9	737.3	2	**0.33**	**0.017**	250
Taskfreak/ *sqlmap*	4	13.0	737.3	0	**1.00**	**0.017**	303
Theorganizer- / *burpsuite*	28	123.0	28.6	0	**1.00**	**0.811**	302
Theorganizer- / *sqlmap*	27	133.0	28.6	0	**1.00**	**0.823**	312
Wordpress-newstatpress- / *burpsuite*	0	5,389.6	9,442.9	4	**0**	**0.363**	306
Wordpress-newstatpress- / *sqlmap*	154	5.562.0	11,586.7	16	**0.91**	**0.324**	294
Wordpress-landingpage- / *sqlmap*	155	5,682.0	13,973.6	159	**0.49**	**0.289**	300
GreenSQL							
HotelRS/ *Xavier*	1,871	0.0	1.039.2	0	**1.0**	**0.000**	6
SugarCRM/ *Xavier*	133	2,020.6	8.047.7	63	**0.68**	**0.201**	5
Taskfreak/ *burpsuite*	3	0.3	749.9	0	**1.0**	**0.000**	6
Taskfreak/ *sqlmap*	4	0.4	749.9	0	**1.0**	**0.001**	5
Theorganizer- / *burpsuite*	28	59.1	92.5	0	**1.0**	**0.390**	5
Theorganizer- / *sqlmap*	27	58.7	102.9	0	**1.0**	**0.363**	5
Wordpress-newstatpress- / *burpsuite*	4	1,372.0	13,460.0	0	**1.0**	**0.093**	5
Wordpress-newstatpress- / *sqlmap*	170	1,360.4	15,788.3	0	**1.0**	**0.079**	5
Wordpress-landingpage- / *sqlmap*	314	1,671.6	17,984.0	0	**1.0**	**0.085**	5

Table 4 shows the Recall, FPR, and execution time of *antiSQLi* and *GreenSQL*. We can observe that FPR and Recall of *antiSQLi* are significantly worse compared to those of SOFIA. *antiSQLi* missed many attacks on all the case studies (FN > 0) and, as a result, Recall of some cases is very low. It wrongly classified safe statements as attacks (high FP rate) on many cases, leading to a poor FPR of 0.823 (or 82.3%). In particular, no attack was correctly identified on one case (TP = 0). *GreenSQL* is as good as SOFIA in all but one subject (*SugarCRM*) with respect to Recall. However, *GreenSQL* reported many false positives that resulted in an order of magnitude higher FPR (up to 0.363) as compared to that of SOFIA (0.006).

Regarding the time taken to process an SQL file, *antiSQLi* takes on average 300ms, which is higher than the average time required by SOFIA (92ms). *GreenSQL* takes only about 5ms and is therefore faster than SOFIA at the cost of a much higher number of false positives. It is worth noticing that *GreenSQL* is a leading industrial tool while SOFIA is a currently research prototype. Besides, because of technical reasons, *GreenSQL* could not run on the HPC but on a server that happened to have a higher CPU frequency than the computer used for SOFIA and *antiSQLi*. This setting clearly favoured *GreenSQL* in detecting attacks faster and prevents us from drawing objective conclusions regarding its comparison with SOFIA regarding its run-time speed.

To compare accuracy and classification time, we used the non-parametric Wilcoxon test. The use of non-parametric tests requires no distributional assumption. Such a test checks whether differences in performance recorded for SOFIA and *antiSQLi* are statistically significant[16]. Results show that SOFIA performs significantly better than *antiSQLi* with respect to Recall, FPR and time (*p-values* are, respectively, 0.028, 0.008 and 0.011). Similarly, SOFIA fares significantly better than *GreenSQL* in terms of FPR (*p-value* = 0.028). Thus, we can provide a clear answer to $\mathbf{RQ}_3$:

SOFIA is significantly more accurate than antiSQLi *and* GreenSQL *and significantly faster than* antiSQLi *in classifying legitimate SQL statements and SQLi attacks.*

5.7 Threats to Validity

To help increase the external validity of our results, which is the main challenge in our study, we relied on various applications from different domains and written using different programming languages, and three different attack generation tools. Further, to ensure our accuracy results were realistic, we resorted to standard 10-fold validation involving multiple training/testing data sets. However, we have to recognise the inherent limitations of such studies as we cannot predict accuracy on future vulnerabilities. The fact that our learning approach does not rely on the specific vulnerabilities in our application systems helps alleviate this problem but does not eliminate it entirely.

6. RELATED WORK

We review SQLi detection techniques that are based on analysing SQL statements at run-time. Table 5 summarizes which of our security oracle requirements (see Section 3.1) are met by related work. Our requirements are: Req_1 training is independent from known attacks; Req_2 classification is neither based on the knowledge of test input data nor on the input data generation algorithm; and Req_3 analysis does not require access to source code.

Table 5: Security oracle requirements met by related work.

Papers	Req_1	Req_2	Req_3
Halfond et al. [11] Bisht et al. [6] Buehrer et al. [7] Kemalis et al. [14]	✓	✓	–
Pinzon et al. [21]	–	✓	✓
Liu et al. [17] Valeur et al. [26]	✓	✓	✓

White-box approaches [11] [6] [7] [14] require the source code to be available for instrumentation or analysis, so they do not meet requirement Req_3. Halfond et al. proposed AMNESIA [11], a method based on program analysis to build models (non-deterministic

[16]We assume a 95% confidence level, so a p-value < 0.05 indicates a statistically significant result.

finite automata) for each and every legitimate query of an application. The application is instrumented and each SQL query sent to the database is validated by finding an accepting path in the automaton. If not possible, the query is considered to be an attack. Bisht et al. proposed CANDID [6], an approach that compares a developer's intended query structure with the actual query structure found during program execution. While both AMNESIA and CANDID show promising evaluation results and they meet requirement Req_1 and Req_2, but they require access to the source code and its instrumentation, which limits their applicability and violates requirement Req_3.

Buehrer et al. have proposed SQLGuard, which compares parse trees of each SQL statement before and after the inclusion of user inputs at runtime [7]. If the trees corresponding to a statement (with and without user inputs) are different after removing constants, then the statement is considered to result from SQLi attacks. In comparison to our approach, which does not require any change to the source code, the application of SQLGuard requires SQL queries to be rewritten using a Java library provided by SQLGuard's authors, thus Req_3 is not fully met.

Several approaches based on anomaly detection have been proposed in the literature [14, 17, 21, 26]. Many of them look similar to ours because they contain the same two high-level steps: training and detection. We provide below a detailed comparison with our work, which aims at addressing three main limitations of practical importance: false positives, the difficulty to obtain a somewhat complete set of of actual and varied attacks for learning purposes, and the need to handle new attack variants.

Kemalis et al. proposed SQL-IDS, a specification-based approach to detect malicious SQL statements [14]. Even if this approach does analyse source code directly, the user has to provide the specification of all benign SQL statements for the application under protection. SQL-IDS monitors the application during runtime and each query that does not comply with the specification is treated as malicious. Req_3 is not fully met by this approach, because it requires precise knowledge of the source code to be manually provided to the tool. In our proposed approach, the user does not have to provide any specification for benign statements, because the safe model is automatically inferred from the learning set.

The remaining approaches are black-box, i.e. they do not require source code, so they meet requirement Req_3. Pinzon et al. proposed an anomaly detection approach combining neural networks and support vector machine to classify SQL queries into benign or malicious statements [21]. In contrast to our approach, they employ supervised machine learning techniques that require a sufficient number of known attack statements. As it is very much driven by what known attacks were fed to the learner, such an approach does not met requirement Req_1 and it might have difficulties recognising new attacks.

To the best of our knowledge, only two approaches [17][26] fully meet all of our three requirements. However, we overcome their limitations by achieving (i) low sensitivity to learning set incompleteness and (ii) a very low false positive rate.

Liu et al. proposed SQLProb [17], a tool that uses string alignment to detect the part of an SQL query that corresponds to user input, by detecting the difference between a new query (requirements Req_2 and Req_3) and all the queries observed at learning time (requirement Req_1). The tool reports an anomaly when the part of the parse tree that corresponds to detected user input contains non-constant leaf nodes (e.g., arithmetic or logic operators). As one might expect, the reliability of this approach is very sensitive to the completeness of learning, that is whether all types of queries are accounted for. Completeness is required to identify correctly the user input in the SQL query. Identifying user inputs is a difficult and error-prone step that could lead to false positives when training is partial. Unfortunately, false positives were not reported by the authors. After investigation, it turned out that the tool was not available and therefore a comparative study was not possible. Nevertheless, our approach does not entail extracting user inputs and is therefore by design more robust. We do not need all types and variants of safe queries to be available at the training phase and, thanks to the distance measure that we adopt, as demonstrated by our empirical study, we obtain accurate results even when we have no guarantee of completeness during training.

Valeur et al. [26] used machine learning to learn relevant characteristics from user inputs of benign SQL queries. In the training phase, several statistical models characterising relevant features, such as character distribution and string length, are learned from attack-free SQL queries (requirement Req_1), in order to capture patterns and ranges of expected values. In the detection phase, a query is intercepted (requirements Req_2 and Req_3) and parsed, and its input values are compared to detect anomalies against models resulting from training queries with identical parse tree structure. This approach requires an exact match between the parse tree to classify and those in the learning set, while we tolerate a degree of difference using tree distance. Moreover, Valeur et al. learn a statistical distributions of values from legitimate SQL statements, while we remove these values and consider only pruned parse trees. These two fundamental differences allow us to dramatically reduce false positives based on results reported by Valeur et al. In our approach, legitimate statements are correctly classified as safe if they belong (i.e., show acceptable distance) to one of the clusters of our model. Our approach is therefore more resilient, as demonstrated by our empirical results.

7. CONCLUSION

Having in mind realistic industrial settings, we elicited three requirements for an ideal security oracle for security testing of SQLi vulnerabilities. We presented a novel approach that satisfies all of these requirements and we implemented it into a tool that we call SOFIA. SOFIA learns a safe model characterising legitimate SQL statements, based on information logged during normal system executions. To do so, SQL statements are parsed and then parse trees are pruned to remove information that is irrelevant to whether an SQL statement is safe or not. Based on their tree-edit distance, similar parse trees are grouped using clustering and the resulting set of clusters is used as a safe models. SOFIA classifies new SQL statements by comparing and contrasting them with these clusters. Executions whose SQL statements do not sufficiently fit into any of the clusters of the safe model are classified as attacks.

We assessed the accuracy of SOFIA as a security oracle with three different attack generation tools on six PHP and Java systems. No attack was missed and the rate of false positives was very low, thus making SOFIA a reliable and cost-effective approach. Further, the classification of SQL statements was on average below 100ms, thus making it possible to execute a large number of test cases within time constraints. Last, SOFIA significantly outperformed two widely used alternative tools in terms of classification accuracy and execution speed for one of the tools.

8. ACKNOWLEDGMENTS

Mariano Ceccato was funded by the FBK Mobility project in 2015. Other authors were supported by the National Research Fund, Luxembourg (grant FNR/P10/03 and FNR4800382). The work was carried out in collaboration with Six payment services.

9. REFERENCES

[1] C. C. Aggarwal and C. K. Reddy. *Data clustering: algorithms and applications*. CRC Press, 2013.

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

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

[4] A. Avancini and M. Ceccato. Security oracle based on tree kernel methods. In *Trustworthy Eternal Systems via Evolving Software, Data and Knowledge*, pages 30–43. Springer, 2013.

[5] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *Software Engineering, IEEE Transactions on*, 41(5):507–525, May 2015.

[6] P. Bisht, P. Madhusudan, and V. Venkatakrishnan. Candid: Dynamic candidate evaluations for automatic prevention of sql injection attacks. *ACM Transactions on Information and System Security (TISSEC)*, 13(2):14, 2010.

[7] G. Buehrer, B. W. Weide, and P. A. Sivilotti. Using parse tree validation to prevent sql injection attacks. In *Proceedings of the 5th international workshop on Software engineering and middleware*, pages 106–113. ACM, 2005.

[8] S. Christey and R. A. Martin. Vulnerability type distributions in cve. Technical report, The MITRE Corporation, 2006.

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

[10] W. Halfond, J. Viegas, and A. Orso. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, volume 1, pages 13–15. IEEE, 2006.

[11] W. G. Halfond and A. Orso. Amnesia: analysis and monitoring for neutralizing sql-injection attacks. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 174–183. ACM, 2005.

[12] P. Institute. The sql injection threat study. Technical report, Ponemon Institute, 2014.

[13] A. K. Jain. Data clustering: 50 years beyond k-means. *Pattern recognition letters*, 31(8):651–666, 2010.

[14] K. Kemalis and T. Tzouramanis. Sql-ids: a specification-based approach for sql-injection detection. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 2153–2158. ACM, 2008.

[15] A. Kieyzun, P. Guo, K. Jayaraman, and M. Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 199 –209, may 2009.

[16] D. A. Kindy and A.-S. K. Pathan. A survey on sql injection: Vulnerabilities, attacks, and prevention techniques. 2011.

[17] A. Liu, Y. Yuan, D. Wijesekera, and A. Stavrou. Sqlprob: A proxy-based architecture towards preventing sql injection attacks. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 2054–2061, New York, NY, USA, 2009. ACM.

[18] C. D. Manning, P. Raghavan, H. Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[19] L. Marinos and A. Sfakianakis. Enisa threat landscape. Technical report, European Network and Information Security Agency, 2012.

[20] C. D. Nguyen, A. Marchetto, and P. Tonella. Combining model-based and combinatorial testing for effective test case generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 100–110, New York, NY, USA, 2012. ACM.

[21] C. I. Pinzón, J. F. De Paz, Á. Herrero, E. Corchado, J. Bajo, and J. M. Corchado. idmas-sql: intrusion detection based on mas to detect and block sql injection through data mining. *Information Sciences*, 231:15–31, 2013.

[22] A. Reynolds, G. Richards, B. de la Iglesia, and V. Rayward-Smith. Clustering rules: A comparison of partitioning and hierarchical clustering algorithms. *Journal of Mathematical Modelling and Algorithms*, 5(4):475–504, 2006.

[23] D. Shasha and K. Zhang. Fast algorithms for the unit cost editing distance between trees. *Journal of algorithms*, 11(4):581–621, 1990.

[24] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *ACM SIGPLAN Notices*, volume 41, pages 372–382. ACM, 2006.

[25] P. Tonella, R. Tiella, and C. D. Nguyen. Interpolated n-grams for model based testing. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 562–572, New York, NY, USA, 2014. ACM.

[26] F. Valeur, D. Mutz, and G. Vigna. A learning-based approach to the detection of sql attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 123–140. Springer, 2005.

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

[28] J. Williams and D. Wichers. Owasp, top 10, the ten most critical web application security risks. Technical report, The Open Web Application Security Project, 2013.