SQLProb: A Proxy-based Architecture towards Preventing SQL Injection Attacks

Anyi Liu
Department of Computer Science
George Mason University
aliu1@gmu.edu

Yi Yuan
Department of Computer Science
George Mason University
yyuan3@gmu.edu

Duminda Wijesekera, Angelos Stavrou
Department of Computer Science
George Mason University
{wijesekera, astavrou}@gmu.edu

ABSTRACT

SQL injection attacks (SQLIAs) consist of maliciously crafted SQL inputs, including *control code*, used against Database-connected Web applications. To curtail the attackers' ability to generate such attacks, we propose an SQL Proxy-based Blocker (SQLProb). SQLProb harnesses the effectiveness and adaptivity of genetic algorithms to dynamically detect and extract users' inputs for undesirable SQL control sequences. Compared to state-of-the-art protection mechanisms, our method does not require any code changes on either the client, the web-server or the back-end database. Rather, our system uses a proxy that seamlessly integrates with existing operational environments offering protection to front-end web servers and back-end databases. To evaluate the overhead and the detection performance of our system, we implemented a prototype of SQLProb which we tested using real SQL attacks. Our experimental results show that we can detect all SQL injection attacks while maintaining very low resource utilization.

Categories and Subject Descriptors

K.6.m [**Management of Computing And Information Systems**]: Miscellaneous—*Security*; K.6.5 [**Management of Computing And Information Systems**]: Security and protection—*Unauthorized access*

Keywords

Information security, SQL injection attack, Intrusion prevention, Intrusion detection

1. INTRODUCTION

SQL injection attacks(SQLIAs) refer to a class of attacks in which an adversary inserts specially crafted control code into the data fields of an SQL query. A successful SQLIA allows the attacker to gain control of the original query, leading to privilege escalation and extraction of unauthorized information from the database [30]. These attacks exploit inadequacies in the user input handing that are sometimes deeply embedded in the program logic [12, 6].

Earlier research has presented many techniques to defend against SQLIAs. Some research is geared towards attempting to *validate user inputs* [18, 19, 20]. Unfortunately, this is appears to be difficult because most existing approaches have little knowledge of the syntactic structure of generated queries, hence some malicious inputs still manage to pass through [3]. Furthermore, input validation cannot offer protection against more sophisticated attacks such as alternate encoding and stored procedure attacks [12].

Another class of *static analysis* solutions statically screens application source to validate every user input before being integrated into a query [10, 11, 35, 8, 7, 9, 6, 17, 5]. These techniques work when application source code is available. In addition, although dynamic prevention techniques [3, 1] require minimal human interaction, they insert extra *metadata* to delimit user inputs that may change the semantics of the original application code. Moreover automatic preservation of *metadata* is almost impossible. Even if these approaches can effectively detect most SQLIAs, they require extra effort to distinguish user input data, using techniques such as tainting or code instrumentation.

Some researches [27, 28] and commercial solutions such as using `PREPARE` statements require the programmer to define the skeleton of an SQL query in order to make the SQL structure unchangeable. These approaches, although providing a robust mechanism to prevent SQL injection attacks, require the programmer to specify the intended query at every query point, requiring a lot of re-engineering.

Like most code injection attacks, SQLIAs exploit the fact that web applications use a common memory space to keep query code and the user input data, thereby inject code as data and execute them as code [25]. Our system, SQLProb (SQL Proxy-based Blocker) extracts user input from the application-generated query, even when the user input data has been *embedded* into the query, and validate them in the context of the generated query's syntactic structure. We validate user inputs by extracting user inputs and aligning them against valid inputs by using and enhancing genetic algorithm.

SQLProb offers several advantages: First, it is a complete black-box approach that does not require modifying

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.
Copyright 2009 ACM 978-1-60558-166-8/09/03 ...$5.00.

application or database code, thereby avoiding the complexity of tainting, learning, or code instrumentation. Second, our input validation technique does not require metadata or learning. Third, our implementation utilizes an off-the-shelf proxy that requires minimal setup complexity. Finally, SQLProb is independent of the programming language used in the web application.

To evaluate our system, we have employed SQLProb to detect a wide categories of SQL injection attacks. We show that SQLProb can prevent sophisticated attacks, such as the alternate encoding attack and the stored procedure attack. Our experimental results demonstrate remarkable effectiveness in detecting all classes of SQL attacks at a reasonable overhead.

The rest of the paper is organized as follows. Section 2 illustrates the SQLIA with a simple web application example. In Section 3, we first define some terminologies, and then present our system overview as well as the detailed steps of detection process. In Section 4, we evaluate the effectiveness of our approach. Section 5 discusses related work and Section 6 concludes this paper.

2. AN ILLUSTRATIVE EXAMPLE

In this section, we present an actual example of an SQLIA. Figure 1 depicts the login page of an online bookstore that allows users to login by providing user name and password. An SQL injection attack occurs when an attacker causes the web application to generate SQL queries that are functionally different from what the user interface programmer intended. For instance, for a database that stores user names and passwords, an attacker may attempt to gain root privileges by manipulating the user name or password string. Let's say the application contains the following code:

```
query = "SELECT * FROM accounts WHERE login='"
        + request.getParameter("login")
        + "' AND password='"
        + request.getParameter("password") + "'";
```

In this code, the web application retrieves user inputs from *login* and *password*, and concatenates these two user inputs into the query. The above code generates a query for the purpose of user authentication. However, if an attacker enters `admin` into the *login* field and `abc' OR '1=1` into the *password* field, the query string becomes the following, whose condition always evaluated to be a logic tautology, hence an attacker can bypass the authentication process, and gain the root privilege.

```
SELECT * FROM accounts WHERE name='admin'
AND password='abc' OR '1=1'
```

In the above case, the password field, which should have only a password string, is replaced with five sub-strings: string "`abc`", logic control keyword "`OR`", "`1`", logical control assignment "`=`", and "`1`". Particularly, the logical control code "`OR`" connects "`password='abc'`" and "`1=1`" to changes the evaluation of the *Where* clause.

3. SYSTEM DESIGN

3.1 SQLProb System Overview

The main system architecture of SQLProb is illustrated in Figure 2. SQLProb has four main components: (1) The

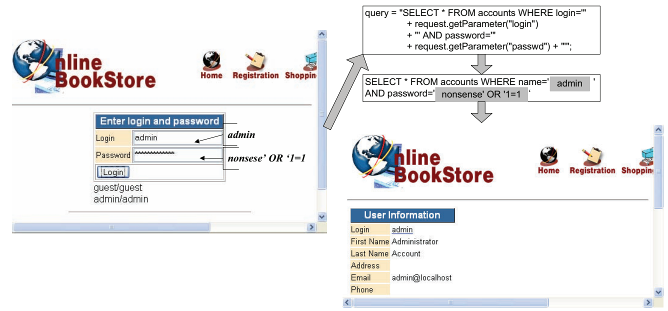


Figure 1: Typical problem overview

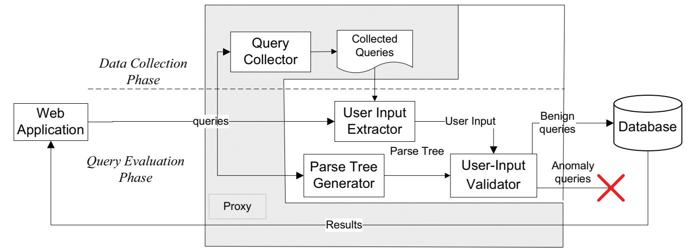


Figure 2: Overview of the SQLProb system architecture

Query Collector processes all possible SQL queries during the data collection phase; (2) The *User Input Extractor* implements a global pairwise alignment algorithm to identify user input data (Section 3.2.1); (3) The *Parse Tree Generator* generates the parse tree for the incoming queries (Section 3.2.2); (4) The *User Input Validator* evaluates weather the user input is normal or malicious based on user input validation algorithms (Section 3.2.3). The shaded area shows the off-the-shelf proxy.

SQLProb uses two phases: the data collection phase and the query evaluation phase, as defined in Section 3.2. During the data collection phase, user inputs validator collects the queries that cover all the functionalities of the application, and stores them in a repository. During the query validation phase, when an application-generated query be captured by the proxy, the proxy forwards it to the user input extractor and the parse tree generator simultaneously. The user input extractor leverages a global alignment algorithm for the application-generated query against the collected query repository, and extracts the user input data. Then, the user input validator validates the extracted user inputs in the parse tree, which is generated by the parse tree generator. If the user inputs are validated to be normal, the generated query will be sent to database directly; otherwise, the query will be discarded as a malicious query. Here we assume that input for the data collection phase is vetted to avoid including existing SQLIAs in our training data. This can easily be done using an automated process.

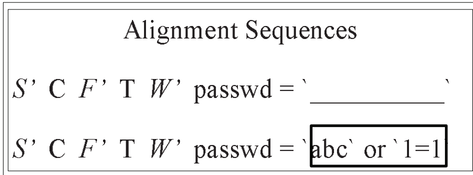
3.2 Terminology

Our terminology is as follows:

- We denote the set of all queries generated by an application $\mathcal{A}$ as $\mathcal{Q} = \{q_i \mid 1 \leq i \leq m\}$. For a query q_i

		S’		C		F’		T		W’		p	a	s	s	w	d	=	‘	a	b	c	‘		o	r		‘	1	=	1	‘
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S’	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	0	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
C	0	1	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
	0	1	2	3	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
F’	0	1	2	3	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
	0	1	2	3	4	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
T	0	1	2	3	4	5	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
	0	1	2	3	4	5	6	7	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
W’	0	1	2	3	4	5	6	7	8	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
	0	1	2	3	4	5	6	7	8	9	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
p	0	1	2	3	4	5	6	7	8	9	10	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
a	0	1	2	3	4	5	6	7	8	9	10	11	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
s	0	1	2	3	4	5	6	7	8	9	10	11	12	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13
s	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	14	14	14	18	14	14	14	14	14	14	14	14	14	14	14	14	14
w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
d	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
=	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17
‘	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	18	18	18	18	18	18	18	18	18	18	18	18	18
a	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	19	19	19	19	19	19	19	19	19	19	19	19
b	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	20	20	20	20	20	20	20	20	20	20	20
c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	21	21	21	21	21	21	21	21	21	21
‘	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	21	21	22	22	22	22	22	22	22

(a) the scoring matrix



(b) The alignment result finalized by Needleman-Wunsch

Figure 3: A complete scoring matrix processed by the Needleman-Wunsch algorithm between two SQL queries and the extracted user input by alignment. In the complete matrix, the shaded cells represent the path, which is traced back from the bottom rightmost cell to the top leftmost cell.

that has $n(n \geq 0)$ user inputs, the set of user inputs is denoted as set $UI(q_i) = \{UI_{i,j} \mid 0 \leq j \leq n\}$. We use the term *user input data* for the *raw* user typed strings and any transformations thereof.

- The *collected queries* of $\mathcal{A}$ is the set $\mathcal{T}(\mathcal{A})$ of all SQL queries generated by $\mathcal{A}$ during the *data collection phase*.
- During the *data evaluation phase*, given a query q_i, the algorithm that compares the similarity for q_i against a query $q_j \in \mathcal{T}(\mathcal{A})$ is given as $Sim(q_i, q_j)$. The query which $\mathring{q}_i$ gives the highest similarity value in $\mathcal{T}(\mathcal{A})$, is called *prototype query* of q_i.
- After incorporating user input, the resulting query string may contains $k(k > 1)$ queries, based on SQL grammar $\mathcal{G}$. The parse tree for a single SQL query q_i is denoted as $Tree(q_i)$. The parse tree of a set of queries $Q = \{q_i \mid 1 < i \leq k\}$ is denoted by $Tree(Q)$.

3.2.1 Separation of User Input

The intuition behind SQLProb is we can pre-generate the structure of all user inputs. Therefore, by having a large enough sample set, it is possible to efficiently compare any user input we receive with one that is in our sample. To efficiently perform this comparison, we use an enhanced version of the Needleman-Wunsch algorithm [31]. This algorithm was originally designed to *globally optimally aligned pairs* of DNA, RNA, or protein sequences.

The Needleman-Wunsch algorithm [31] iteratively constructs a $(N+1)\times(M+1)$ dimensional *scoring matrix* where N and M are the lengths of the two sequences. The algorithm uses four steps: (1) *scoring similarity*, (2) *summing*, (3) *back-tracking*, and (4) *finalization*.

Given two SQL query strings $q_1 = x_1x_2\ldots x_n$ and $q_2 = y_1y_2\ldots y_m$, we can insert gaps, if necessary, to achieve a global maximum alignment between them. In the first two steps, our algorithm first computes the similarity score for each cell of the scoring matrix based on a predefined similarity matrix. The similarity and gap penalty can be defined as a part of the scoring matrix, or can be specified explicitly if otherwise. In our work, we assign 1 for a syntactic match, 0 for a syntactic mismatch, and 5 for a gap. The value of the maximum alignment between q_1 and q_2, or $Sim(q_1, q_2)$ is defined as the sum of terms for each aligned pair of letters $\langle x_i, y_j\rangle$ within the sequences (representing similarity as $s(x_i, y_j)$), plus terms for each gap (representing a penalty as p).

The cell $\mathcal{M}(i, j)$ is the score of the best alignment between the initial segment $x_1x_2...x_n$ of x up to x_i and the initial segment $y_1y_2...y_m$ of y up to y_j. Initially, $\mathcal{M}(0, 0) = 0$, $\mathcal{M}(i, 0) = -ip$, $\mathcal{M}(0, j) = -jp$. Starting with the top leftmost cell $\mathcal{M}(1, 1)$ to bottom rightmost cell $\mathcal{M}(N+1, M+1)$, based on the following equation is used to iteratively fill in in the matrix:

$$y = MAX \begin{cases} \mathcal{M}(i-1, j-1) + s(x_i, y_j) & i \geq 1, j \geq 1 \\ \mathcal{M}(i-1, j) - p & i \geq 1 \\ \mathcal{M}(i, j-1) - p & j \geq 1 \end{cases} \tag{1}$$

After computing the scoring matrix, in the third step, the algorithm *backtracks* from the cell with the highest score(the bottom rightmost cell) according to the following three rules, in order to maximize the alignment score back to the cell with the lowest score (the top leftmost cell) of the matrix.

- If move *diagonal*, do nothing;
- If move *left*, insert a gap into the second sequence;
- If move *up*, insert a gap into the first sequence.

The purpose of backtracking is to access the left, upper, and diagonal cell and move to the cell with the highest score. If all three cells are equal, backtracking will move to the diagonal cell. As illustrated in Figure 3(a), backtracking starts from the cell with the highest score 22(the bottom rightmost cell). The shaded path is traced back to the top leftmost cell, which has the lowest score of 0.

We use the example in Figure 3(b) to demonstrate alignment between two SQL queries. The first query is in the collected query set $\mathcal{T}(\mathcal{A})$ during the *data collection* phase. The second query is generated by the web application at the *data evaluation* phase. In this example, all SQL keywords are abbreviated by their initials. For example, `SELECT` is abbreviated as `S’`, `FROM` is abbreviated as `F’`, and so on. To achieve global alignment, the Needleman-Wunsch algorithm inserts "_", as a gap, into the first query after `password=’`. Therefore, the alignment result for the password is "____________", which indicates that the user's input string for the second query has a length of 12. Accordingly, the sub-string at the corresponding position in the second query is `abc’ OR ’1=1`. Therefore, we can easily extract the user's input for the password in the second query as "abc’ OR ’1=1".

Applications	Before optimization	After optimization	% of reduction
Bookstore	213	55	25.8%
Classifield	395	31	8.1%
EmployDir	218	33	15.1%
Events	364	25	6.8%
Portal	504	56	11.1%

Table 1: The result of optimization

3.2.2 Complexity Analysis and Optimization

To allocate the *prototype query* $\mathring{q}_i$ of a application-generated query q_i, every application-generated query must align with all collected queries during the *data collection* phase. In the worst case, the incoming query q_i must align with all m collected queries in $\mathcal{T}(\mathcal{A})$. Because both time and space complexity of Needleman-Wunsch algorithm are *quadratic*, the overall time complexity to allocate the prototype query is $O(n^3)$, while the space complexity is still $O(n^2)$. Normally, m is much larger than the length of most query strings, making most alignment operations superfluous. The purpose of our optimization is to reduce m to $m'(m' << m)$, such that the overall time complexity will be reduced to approximate $O(n^2)$. To avoid unnecessary alignment operations, we cluster collected queries in the following steps: First, we cluster the collected queries based on different query types. For example, `SELECT` statements and `UPDATE` statements will be categorized into two different clusters. Secondly, within each cluster, queries carrying redundant information will be further aggregated. Particularly, identical queries contains the same query structure, except the user input, will be aggregated. In the aggregated representation, instead of replacing *name* and *password* by the wild-card tokens [22], we fully eliminate the user input strings. Eventually, queries with the identical query structure but different user input strings will be aggregated into the same cluster.

Table 1 demonstrates the number of queries, obtained during the data collection phase, before and after optimization. For different applications, the optimization reduces the size of the collected query, ranging from 6% to 25%.

3.2.3 User Input Validation Algorithms

One of the advantages of our approach is that we can evaluate the extracted user input data in the context of the syntactic structure of the query. In order to motivate our definition of the user input validation algorithm, we return to the two queries(normal and malicious) in Section 2, and show the parse trees for their *where* clause in Figure 4.

Figure 4(a) represents the partial parse tree for the *where* clause of the normal query in section 2, every user input string can find a non-leaf node in the parse tree, such that its sub-tree leaf nodes comprise the entire user input string. Namely, for user input leaf nodes, it is impossible to find a non-leaf node whose decedent leaf nodes contains not only the user input leaf nodes, but also other control leaf nodes. For example, the non-leaf node *ID* for *john* and *ID* for *nonsense*. Both are shown as shaded double octagon. Consequently, our algorithm is given in Table 2.

An example application of the validation algorithm is given in Figure 4(b). The password field is parsed into the set $\bigcup_{i=1}^{n}(leaf(u_i))$ with five leaf nodes: `nonsense`, `OR`, `1`, `=`, and

Data: Parse Tree $Tree(q_i)$ and the set of user input $UI(q_i)$
Result: *True* if the query is an SQLIA, or *False* if otherwise
1. for every user input $UI_{i,j}$ in $UI(q_i)$
 2. do *depth-first-search* upward from every leaf node $leaf(u_i)$ parsed from $UI_{i,j}$, according to SQL grammar $\mathcal{G}$;
 3. Searching stops when all the searching path intersect at a non-leaf node *nl_node*;
 4. do *breath-first-search* downward from *nl_node* **until** reaching all m leaf nodes $leaf(node)_k$;
 5. if $\bigcup_{i=1}^{n}(leaf(u_i)) \subset \bigcup_{k=1}^{m}(leaf(node)_k)$ **then**
  Return *True*;
  else Return *False*;
 end
end

Table 2: The validation algorithm

Type	Attack Description	Detected?
Tautology	Injecting one or more conditional statement	Yes
Logically Incorrect Queries	Information gathering, extract data	Yes
Union Queries	Return data from a different table	Yes
Piggy-Backed Queries	New queries piggy-back on the original	Yes
Stored Procedures	Invoking stored procedure	Yes
Inferences	Infer answers from apps' response	Yes
Alternate Encoding	Injecting modified control text	Yes

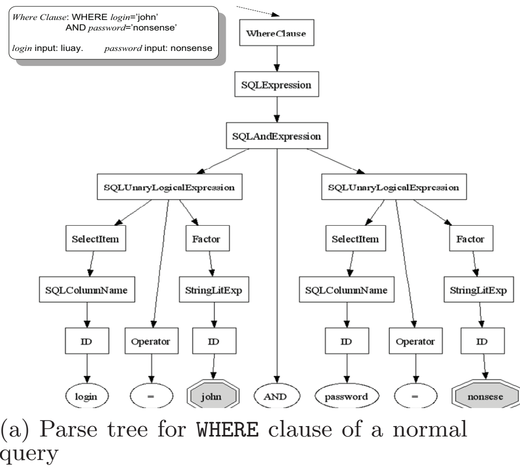
Table 3: Different Categories of Attacks used in effectiveness evaluation

`1`. Next, we do depth-first-search from these five leaf nodes. The traversed paths intersect at a non-leaf node, *SQLExpression*. Finally, we do breath-first-search from *SQLExpression* to reach all the leaf nodes of the parse tree, which is a superset of $\bigcup_{i=1}^{n}(leaf(u_i))$, implying that the input string u_i is malicious.

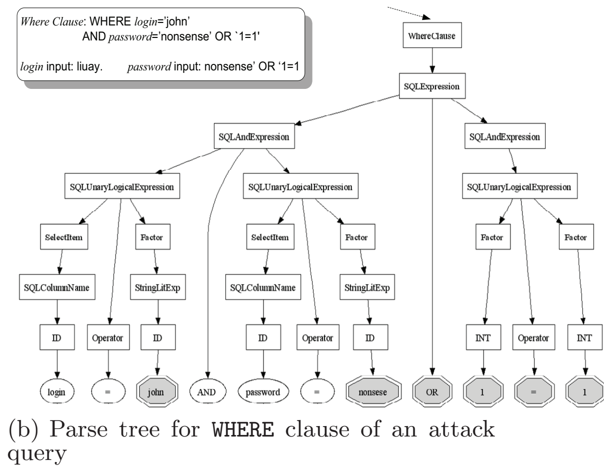
The algorithm described above takes quadratic time, because step 2 and step 3 take time of $n \times h$, where n is the number of leaf nodes parsed by u_i, and h is the average number of layers from leaf nodes to *nl_node* in the parse tree. In addition, step 4 takes time complexity for a *breath-first-search* is $O(n^2)$. Therefore, the overall time complexity is $O(n^2)$.

4. EVALUATION

To measure the overhead and performance of our approach, we used a prototype implementation of *SQLProb*. The current version of *SQLProb* is implemented by Java and tested on a Virtual Machine with 1 GB RAM running Fedora 9. We use *MySQL 5.0.27* as the back-end database server. For every SQL query initiated by a web application, we use a customized *MySql Proxy* [33] to collect it, and determine if it is benign or malicious, before sending it to the database. If the query is determined as benign, the query will be forwarded to the database; otherwise, it will be dropped imme-



(a) Parse tree for WHERE clause of a normal query



(b) Parse tree for WHERE clause of an attack query

Figure 4: The parse tree of a normal *where* clause (left) and an attack one(right). The shaded double octagons represent the leaf nodes parse from user inputs .

Application Name	No of Requests	LOC	Failure & Syntax. Error
Portal	7483	16,453	2999
Bookstore	6492	16,959	1612
Classifieds	6544	10,949	1475
EmplDir	7038	5,658	1994
Events	7109	7,242	2240

Table 4: Applications from the Amnesia

diately. To minimize the network latency, we use *wget 1.10* [39] to replay HTTP request from a different machine within the same Ethernet subnet to the machine, which runs both web application server and the prevention engine.

JavaCC [37] was used to automate parse tree generation process. Specifically, we used *JJTree* [38], the pre-processor of JavaCC, to generate parse trees. Figure 5 illustrates the screen-shot of the web application and the input of *login* and *password*. The corresponding parse tree is illustrated on the right hand side of the figure. The source code of the vulnerable web application are public available from *http://www.gotocode.com*.

4.1 Experimental Setup

We use Amnesia attack test suite [17], containing both benign and attacking string patterns. The attacking result has been extensively explored before (such as in [5, 28]). Although the test suite contains 30 different attack patterns and the malicious codes have been injected successfully, we noticed that the set of attack patterns may not be complete. To ensure the test suite is as complete as possible, we further extended the attack pattern by including a wide category of the real-world attacking patterns[12], in order to guarantee that the malicious attacking string patterns return "sensitive" information. Table 3 illustrates a list of vulnerabilities, as well as injection attacks exploiting those vulnerabilities. Those vulnerabilities and attacks cover the most known SQLIA scenarios; furthermore, the combination of those can come up with more complicated new attacks. Table 4 summarizes the characteristics of Amnesia test suite. The second column lists the number of web requests, and the

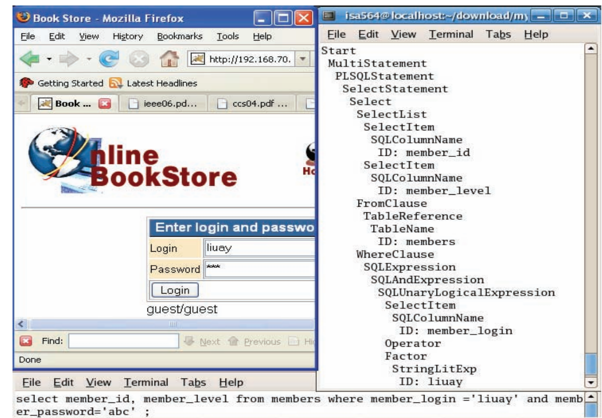


Figure 5: The screen-shot of SQLProb

third column lists the lines of code (LOC) for each application. Since the test suite contains a large number of web requests that resulted in invalid SQL queries, the fourth column reports the number of invalidate web requests.

4.2 Detection & Resource Overhead

The objective of the first set of experiments is to demonstrate the effectiveness of the proposed technique to prevent SQL injection attacks. We ran the the attack suite and detect by SQLProb, which can achieve 100% detection rate for all the attacks in the test suite.

The objective of the second set of experiments is to evaluate the performance of SQLProb. The first performance metric is the response time per web request. For each web request sent to the application, we measure the web application's original response time, the response time only with proxy, and the response time with SQLProb. All the results have 95% confidence interval, which are shown in Figure 6. Clearly, for different application, the proxy only introduces reasonable delay, which ranges from 16.7%(`Portal`) to 25.7% (`Events`). For every request, the prevention engine had varying delays ranging from 59.5%(`Empldir`) to

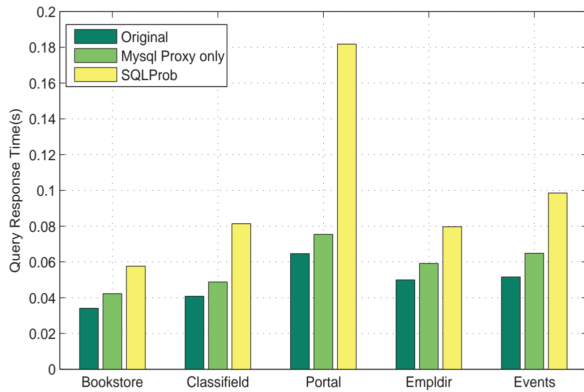


Figure 6: The response time

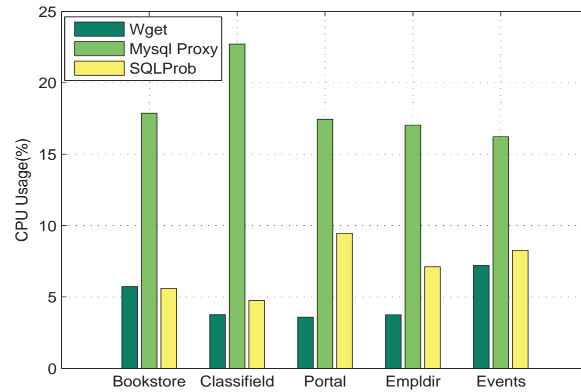


Figure 7: The CPU usage

181.3%(`Portal`). The delay of our prevention engine mainly attributed to query alignment, parse tree generation, and user input validation procedure e.g., most queries collected by `Portal` at training phase are very long strings. Due to the nature of Needleman-Wunsch algorithm, the incoming query must take longer time to align with the collected long queries in order to determine its prototype query. Refining optimization to reduce the alignment time comprises an interesting future direction.

The second performance metric is the resource usage, such as CPU usage. Figure 7 demonstrates the CPU usage over time for the web applications. All the results have 95% confidence interval. The results clearly demonstrate that SQLProb demands much less computational resource than *Mysql Proxy*. It only takes 20.9% to 54.2% CPU usage of which taken by *Mysql Proxy*. Clearly, SQLProb works faster than Mysql Proxy in processing the incoming queries.

5. RELATED WORK

SQL injection attacks have researched in depth, resulting in a number of protection techniques that can be broadly categorized as: *input validation*, *static analysis* , *learning-based prevention*, and *dynamic prevention* approaches. We compare ours with each of these categories.

Input Validation Because the root cause of SQLIAs is the intermingling of data and control code, improper input validation accounts for most security problems in database and web applications. Many input validation approaches are signature-based, resulting in incomplete input validation routines introducing false alerts. In [19, 20], a human-developed security- policy description language(SPDL) specifies and enforces user input constraints by analyzing and transforming HTTP requests/responses to enforce the specified policy. This approach is human-based, and requires developer to know which data and pattern filter to apply to the data. PowerForms [18] and Commercial tools, such as AppShield [32] and InterDo [35] provide the similar methodology.

The common weakness of these techniques are: they have no insight on the structure of the generated queries, and therefore, may still admit bad inputs. In addition, they ignore the fact that the original user input may subject to manipulation and transformation, which may eventually defeat the effectiveness of this approach. Our approach is complimentary to most of the existing input validation approaches. In addition, we avoid the complicated steps to trace the user input string manipulation throughout the application. Furthermore, compare with some products claim to have the *proxy-alike* capability of intercepting all incoming queries and blocking the suspicious ones [42, 40, 41], our approach does not require specifying any signature or rule for known attacks.

Static Analysis To guarantee security, [6, 7, 8, 26] perform static analysis over the entire application's source code to ensure that every piece of input is subject to an input validation check before being incorporated to a query. However, this approach requires the entire source code of the application, while our approach is a black-box based approach that requires no source code for applications and databases.

Our work is closely related to the recent work of CANDID [5], which dynamically mines programmer intended query structure on any user input. While it is an effective approach, it requires extra instrumentation to transform the web application code, usually tied to a specific programming language. Moreover, there is no guarantee that the byte code transformation process is error-free, and will not introduce any potential vulnerabilities. Furthermore, byte code transformation is expensive and may negatively impact the availability of the web applications.

Learning-based Prevention A set of learning-based approaches have been proposed to learn all the intended query structure statically [17] or dynamically [2, 22]. The effectiveness of detection largely depends on the accuracy of the learning algorithms. Comparing with this category of approaches, our approach demonstrates at least two advantages. First, our approach neither *require* any learning algorithm, nor *limited to* the number of collected queries. Second, our approach validates the user input within the syntactic structure of generated query, which more efficiently reveals the syntactic meaning of the user input.

Dynamic Prevention Dynamic tainting approaches [14, 16, 15] taint the input strings and track those taints along the information flow of a program. All of them require not only the source code of the entire application, but also the collaboration of external libraries.

Many recent work [3, 1, 4], explicitly mark the user input data by using *metadata*. Although they all indeed work efficiently, it is widely believed that the metadata introduces many disadvantages, such as changing the semantics of the original program, and requiring metadata preservation functions throughout the application, which is almost impossible in an automatic manner [5]. In contrast, our approach is a complete black-box approach that requires no source code of web applications. SQLrand [13] leverages secret keys, while SMask [24] uses keyword mask to randomize and de-randomizes every SQL keyword through a proxy filter before passing the query to the database. Thus, the injected commands will cause a syntactic failure after passing to the proxy filter. This approach has immediate drawbacks: SQLrand requires extra efforts to *rewrite* all the "plain-text" queries in the web applications to the "randomized" ones. In addition, the security of the above approaches depends on the secret key, which is possibly be compromised by brute force attacks. Furthermore, this technique "decrypt" SQL instructions throughout the applications, which imposes tremendous overheads. In comparison, our approach does not require any secret key. Our approach has a different purpose by using the proxies for the alignment purpose.

6. CONCLUSION

We have presented SQLProb, a novel online and adaptive detection system against SQLIAs. SQLProb employs dynamically user input extraction and analysis taking into consideration the context of query's syntactic structure. Unlike current protection techniques, our approach is fully modular and does not require access to the source code of the web applications or the database. In addition, our system is easily deployable to existing enterprise environments and can protect multiple front-end web applications without any modifications. To measure the performance and overhead of our technique, we developed a prototype of SQLProb. Our experimental results indicate that we can achieve have high detection rate with reasonable performance overhead making our system ideal for environments where software or architecture changes is not an economically viable option.

7. ACKNOWLEDGEMENTS

We thank William Halfond and Alex Orso for providing SQL injection application testbed. We would also like to thank Xuxian Jiang for providing useful comments on an early version of the paper. Finally, we thank anonymous reviewers for their insightful comments and suggestions.

8. REFERENCES

[1] T. Pietraszek, C. Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection(RAID)*, pages 124-145, 2005.

[2] F. Valeur, D. Mutz, and G. Vigna. A learning-based approach to the detection of sql attacks. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 123140, 2005.

[3] Z. Su, and G. Wassermann. The essence of command injection attacks in web applications. In *Proceedings of the Symposium on Principles of Programming Languages(POPL)*, pages 372382, 2006.

[4] G. Buehrer, B. W. Weide, and P. Sivilotti, Using parse tree validation to prevent sql injection attacks. In *Proceedings of the Fifth International Workshop on Software Engineering and Middleware(SEM)*, 2005.

[5] S. Bandhakavi, P. Bisht, P. Madhusudan, and V.N. Venkatakrishnan. CANDID: preventing sql injection attacks using dynamic candidate evaluations. In *Proceedings of the 14th ACM Conference on Computer and Communications Security(CCS)* , pages 12-24, 2007.

[6] Y. Xie, and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th USENIX Security Symposium*, pages 179192, 2006.

[7] V. Livshits, and M. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.

[8] M. Lam, J. Whaley, V. Livshits, M. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the ACM Principles of Database Systems(PODS)*, June 2005.

[9] M. Martin, V. Livshits, and M. Lam. Finding application errors using PQL: a program query language. In *Proceedings of the 20th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, San Diego, California, October 2005.

[10] D. Scott and R. Sharp. Abstracting application-level web security. In *Proceedings of the World Wide Web(WWW)*, pages 396407, 2002.

[11] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the Thirteenth International World Wide Web(WWW)*, pages 40-52, New York, May 2004.

[12] W. Halfond, J. Viegas, and A. Orso. A Classification of SQL injection attacks and countermeasures. In *Proceedings of the International Symposium on Secure Software Engineering(SEEE)*, March 2006.

[13] S. Boyd and A. Keromytis. SQLrand: Preventing SQL injection attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security(ACNS)*, pages 292302, 2004.

[14] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *Proceedings of the 20th IFIP International Information Security Conference(SEC)*, pages 295-308, 2005.

[15] W. Halfond, A. Orso. and P. Manolios. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *Proceedings of the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 175185, 2006.

[16] W. Xu, . S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, pages 121136, 2006.

[17] W. Halfond, and A. Orso. AMNESIA: analysis and

monitoring for NEutralizing SQL-injection attacks, n *Proceedings of the IEEE and ACM International Conference on Automated Software Engineering(ASE)*, 2005.
[18] C. Brabrand, A. Mller, M. Ricky, and M. I. Schwartzbach. Powerforms: Declarative client-side form field validation. In *Proceedings of the World Wide Web(WWW)*, 2000.
[19] D. Scott and R. Sharp. Abstracting application-level web security. In *Proceedings of the World Wide Web(WWW)*, 2002.
[20] D. Scott and R. Sharp. Specifying and enforcing application-level web security policies. In *IEEE Transactions in Knowledge and Data Engineering (TKDE)*, 15(4):771.783, 2003.
[21] D. Balzarotti, M. Cova, V. Felmetsger and G. Vigna. Multi-module vulnerability analysis of web-based applications. In *Proceedings of the 14th ACM Conference on Computer and Communications Security(CCS)*, pages 25-35, 2007.
[22] S. Lee, W. Low, P. Wong. Learning fingerprints for a database intrusion detection system. In *Proceedings of the 7th European Symposium on Research in Computer Security(ESORICS)*, pages 264-280, 2002.
[23] Y. Kosuga, K. Kono, M. Hanaoka, M. Hishiyama, Y. Takahama. Sania: Syntactic and semantic analysis for automated testing against SQL injection. In *Proceedings of the 23rd Annual Computer Security Applications Conference(ACSAC)*, pages 107-117, 2007.
[24] M. Johns, C. Beyerlein. SMask: Preventing Injection Attacks in Web Applications by Approximating Automatic Data/Code Separation. In *Proceedings of the 22nd ACM Symposium on Applied Computing (SAC)*, Seoul, Korea, March 2007.
[25] R. Riley, X. Jiang, D. Xu. An Architectural Approach to Preventing Code Injection Attacks. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks(DSN)*, pages 30-40, 2007.
[26] C. Gould, Z. Su, and P. Devanbu. JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pages 697698, 2004.
[27] W. Cook, and S. Rai. Safe query objects: statically typed objects as remotely executable queries. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 97106, 2005.
[28] Y. Kosuga, K. Kono, M. Hanaoka, M. Hishiyama, and Y. Takahama. Sania: Syntactic and Semantic Analysis for Automated Testing against SQL Injection. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC '07)*, Miami Beach, Florida , 2007.
[29] X. Jiang, and D. Xu: Profiling Self-Propagating Worms via Behavioral Footprinting, In *Proceedings of the 4th ACM Workshop on Recurring Malcode*, Fairfax, VA, November 2006
[30] T.O.Fundation. Top ten most critical web application vulnerabilities, 2005. `http://www.owasp.org/documentation/topten.html`.
[31] R. Durbin, S. Eddy, and A. Krogh. Biological sequence analysis. Cambridge University Press, ISBN: 0521629713, 1998.
[32] Sanctum Inc. AppShield 4.0 Whitepaper, 2002. `http://www.sanctuminc.com`.
[33] MySQL Proxy Project Wiki. `http://forge.mysql.com/wiki/MySQL_Proxy`.
[34] SPI Dynamics. Web application security assessment. SPI Dynamics Whitepaper, 2003.
[35] Kavado, Inc. InterDo Vers. 3.0, 2003.
[36] SQLBrute - SQL Injection brute force tool. `http://www.darknet.org.uk/2007/06/sqlbrute-sql-injection-brute-force-tool/`.
[37] JavaCC Project. `https://javacc.dev.java.net/`.
[38] JJTree. `https://javacc.dev.java.net/doc/JJTree.html`.
[39] Wget `http://ftp.gnu.org/gnu/wget/`.
[40] McAfee Entercept Database Edition. `http://www.anidirect.com/products/intrusionprevention/ds_entercept_databaseedition.pdf`.
[41] GreenSQL. `http://www.greensql.net/`.
[42] SANA Security's Primary Response. `http://www.sanasecurity.com/common/files/PR3.0_datasheet.pdf`.