AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks

William G.J. Halfond and Alessandro Orso
College of Computing
Georgia Institute of Technology
{whalfond|orso}@cc.gatech.edu

ABSTRACT

The use of web applications has become increasingly popular in our routine activities, such as reading the news, paying bills, and shopping on-line. As the availability of these services grows, we are witnessing an increase in the number and sophistication of attacks that target them. In particular, SQL injection, a class of code-injection attacks in which specially crafted input strings result in illegal queries to a database, has become one of the most serious threats to web applications. In this paper we present and evaluate a new technique for detecting and preventing SQL injection attacks. Our technique uses a model-based approach to detect illegal queries before they are executed on the database. In its static part, the technique uses program analysis to automatically build a model of the legitimate queries that could be generated by the application. In its dynamic part, the technique uses runtime monitoring to inspect the dynamically-generated queries and check them against the statically-built model. We developed a tool, AMNESIA, that implements our technique and used the tool to evaluate the technique on seven web applications. In the evaluation we targeted the subject applications with a large number of both legitimate and malicious inputs and measured how many attacks our technique detected and prevented. The results of the study show that our technique was able to stop all of the attempted attacks without generating any false positives.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging—*Monitors*;

General Terms: Security, Verification

Keywords: SQL injection, static analysis, runtime monitoring

1. INTRODUCTION

Many organizations have a need to store sensitive information, such as customer records or private documents, and make this information available over the network. For this reason, database-driven web applications have become widely deployed in enterprise systems and on the Internet. Along with their growing deployment, there has been a surge in attacks that target these applications. One type of attack in particular, *SQL Injection Attacks* (*SQLIAs*), is especially harmful. SQLIAs can give attackers direct access to the database underlying an application and allow them to leak confidential, or even sensitive, information. There are many examples of SQLIAs with serious consequences, and the list of victims of such attacks includes high-profile companies and associations, such as Travelocity, FTD.com, Creditcards.com, Tower Records, and RIAA. Even more alarming is a study performed by the Gartner Group on over 300 web sites, which found that 97% of the sites audited were vulnerable to this kind of web attack. In fact, SQLIAs have been described as one of the most serious security threats to web applications [2, 21].

SQL injection refers to a class of code-injection attacks in which data provided by the user is included in a SQL query in such a way that part of the user's input is treated as SQL code. SQLIAs are a type of vulnerability that is ultimately caused by insufficient input validation—they occur when data provided by the user is not properly validated and is included directly in a SQL query. By leveraging these vulnerabilities, an attacker can submit SQL commands directly to the database. This kind of vulnerability represents a serious threat to any web application that reads input from the users (e.g., through web forms or web APIs) and uses it to make SQL queries to an underlying database. Most web applications used on the Internet or within enterprises work this way and could therefore be vulnerable to SQL injection.

Although the vulnerabilities that lead to SQLIAs are well understood, they persist because of a lack of effective techniques for detecting and preventing them. Programming practices such as defensive programming and sophisticated input validation techniques can prevent some vulnerabilities. However, attackers continue to find new exploits that can avoid the checks programmers put in place (e.g., [16, 19, 24]). Moreover, defensive programming is labor-intensive, which makes it an impractical technique for protecting large legacy systems. General tools such as firewalls and current Intrusion Detection Systems (IDSs) are also typically ineffective against SQLIAs—SQLIAs are performed through ports used for regular web traffic (usually open in firewalls) and work at the application level (unlike most IDSs). Finally, most analysis-based techniques for vulnerability detection do not address the specific characteristics of SQLIAs and are thus ineffective in this context. The few analysis techniques specifically designed to target SQLIAs provide only partial solutions to the problem. In particular, dynamic techniques, such as penetration testing, introduce issues of completeness and often result in false negatives being produced, whereas techniques based on static analysis are either too imprecise or only focus on a specific aspect of the problem.

In this paper, we propose a novel technique to counter SQLIAs.[1] Our technique builds upon work done in model-based security and in program analysis and uses a combination of static and dynamic analysis techniques that is specifically designed to target SQLIAs. The key insights behind the development of the technique are that (1) the information needed to predict the possible structure of the queries generated by a web application is contained within the application's code, and (2) an SQLIA, by injecting additional SQL statements into a query, would violate that structure. Therefore, our technique first uses static program analysis to analyze the application code and automatically build a model of the legitimate queries that could be generated by the application. Then, at runtime, the technique monitors all dynamically-generated queries and checks them for compliance with the statically-generated model. Queries that violate the model are classified as illegal, prevented from executing on the database, and reported to the application developers and administrators.

In the paper, we also present an empirical evaluation of the technique. We implemented the technique in a prototype tool, AMNESIA, and used the tool to evaluate the technique on a set of seven subjects of various types and sizes. We targeted the subjects with a large number of both legitimate accesses and SQL-injection attacks and assessed the ability of our technique to detect and prevent the attacks without stopping any legitimate access to the database. The results of the evaluation are very promising. AMNESIA was able to stop all of the 1,470 attacks without generating any false positive for the 3,500 legitimate accesses. Moreover, our technique proved to be very efficient, in that it imposed a negligible overhead on the subject web applications.

The main contributions of this work are:

- The presentation of a new technique to counter SQL-injection attacks that combines static analysis and runtime monitoring.
- A tool, AMNESIA, that implements the technique for Java-based web applications.
- An empirical evaluation of the technique that shows the effectiveness and the efficiency of the technique.
- The development of a test bed for the evaluation of web-application protection techniques that can be reused by other researchers.

The rest of this paper is organized as follows. In Section 2 we review and discuss related work. We provide an example of an SQLIA in Section 3 and discuss our technique in Section 4. We examine additional considerations in Section 5. Section 6 presents our empirical evaluation and, finally, we conclude and outline future work in Section 7.

2. RELATED WORK

Many existing techniques, such as filtering, information-flow analysis, penetration testing, and defensive coding, can detect and prevent a subset of the vulnerabilities that lead to SQLIAs. In this section, we list the most relevant techniques and discuss their limitations with relation to SQLIAs.

Defensive Programming. Developers have employed a range of code-based solutions to counter SQLIAs. Input validation based techniques include checking user input for keywords, identifying known malicious patterns, and escaping potentially troublesome characters. While these techniques can stop straightforward and unsophisticated attacks, attackers have learned to use alternate encoding schemes such as hexadecimal, ASCII and Unicode to obfuscate their attacks. Furthermore, simply checking user input for malicious keywords would clearly result in a high rate of false positives, since an input field could legally contain words that match SQL keywords (i.e. "FROM", "OR", "AND"). Another widely proposed coding solution is to use stored procedures for database access. The ability of stored procedures to prevent SQLIAs is dependent on its implementation. The mere fact of using stored procedures does not protect against SQLIA. Interested readers may refer to [1, 11] for examples of how applications that use stored procedures, escaping of characters, and different forms of input validation can be vulnerable to SQLIAs.

Two recent approaches, SQL DOM [18] and Safe Query Objects [6], use encapsulation of database queries to provide a safe and reliable way to access databases. These techniques offer an effective way to avoid the SQLIA problem by changing the query-building process from an unregulated one that uses string concatenation to a systematic one that uses a type-checked API. (In this sense, SQL DOM and Safe Query Objects can be considered instances of defensive coding.) Although effective, these techniques have the drawback that they require developers to learn and use a new programming paradigm or query-development process, unlike our technique.

In general, defensive coding has not been successful in completely preventing SQLIA (e.g., [16, 19, 24]). Attackers keep finding new attack strings or subtle variations on old attacks that can avoid the checks programmers put in place. While improved coding practices (e.g., [11]) can help mitigate the problem, they are limited by the developer's ability to generate appropriate input validation code and recognize all situations in which it is needed. Our approach, being fully automated, can provide stronger guarantees about the completeness and accuracy of the protections put in place.

General Techniques Against SQLIAs. Other researchers have developed techniques specifically targeted at SQLIAs. Scott and Sharp [23] use a proxy to filter input and output data streams for a web application based on policy rules defined at the enterprise level. Although this technique can be effective against SQLIA, it requires developers to correctly specify filtering rules for each application input. This step of the process is prone to human-error and leaves the application vulnerable if the developer has not adequately identified all injection points and correctly expressed the filtering rules. Like defensive coding practices, this technique cannot provide guarantees of completeness and accuracy.

Huang and colleagues [12] propose WAVES, a black-box technique for testing web applications for SQL-injection vulnerabilities. This technique improves over general penetration-testing techniques by using machine learning to guide its testing, but like all black-box testing techniques, it cannot provide guarantees of completeness that static analysis based techniques are able to provide.

Boyd and colleagues propose SQLrand, an approach based on randomization of SQL instructions using a key [3], which extends a previous approach to counter general code-injections attacks [14]. In this approach, SQL code injected by an attacker would result in a syntactically incorrect query. Although effective, this technique could be circumvented if the key used for the randomization were exposed. Moreover, the approach imposes a significant overhead in terms of infrastructure because it requires the integration of a special proxy in the web-application infrastructure.

Two related approaches by Nguyen-Tuong and colleagues [20] and Pietraszek and Berghe [22] modify the PHP interpreter to track precise taint information about user input. The techniques use a

[1]An early version of this work is described in a paper presented at WODA 2005 [10].

context sensitive analysis to detect and reject queries if untrusted input has been used to create certain types of SQL tokens.

Valeur and colleagues [25] propose the use of an Intrusion Detection System (IDS) to detect SQLIA. Their IDS system is based on a machine learning technique that is trained using a set of typical application queries. The technique builds models of the typical queries and then monitors the application at runtime to identify queries that do not match the model. Overall, this technique uses an approach similar to ours, in that it builds expected query models and then checks dynamically-generated queries for compliance with the model. Their technique, however, like most techniques based on learning, can generate large number of false positive in the absence of an optimal training set.

Static Analysis Techniques. The Java String Analysis (JSA) library, developed by Christensen, Møller, and Schwartzbach [5] provides us with a mechanism for generating models of Java strings. JSA performs a conservative string analysis of an application and creates automata that express all the possible values a specific string can have at a given point in the application. Although this technique is not directly related to SQLIA, it is important to our work because we use the library to generate intermediate forms of of our SQL-query models. Other works such as JDBC-Checker [7, 8], also make use of this library in their analysis.

JDBC-Checker is a technique for statically checking the type correctness of dynamically-generated SQL queries by Gould, Su, and Devanbu [7, 8]. This technique was not intended to detect and prevent general SQLIAs, but can nevertheless be used to prevent attacks that take advantage of type mismatches in a dynamically generated query string to crash the underlying database. JDBC-Checker is able to detect one of the root causes of SQLIA vulnerabilities in code, which is improper type checking of input. However, this technique would not catch more general forms of SQLIAs because these attacks must generate syntactically and type correct queries in order to be successful. This technique also relies on the JSA [5] library, and we use a similar approach to build an intermediate form of our SQL-query models (see Section 4.2).

Wassermann and Su propose an approach that uses static analysis combined with automated reasoning to verify that the SQL queries generated in the application layer cannot contain a tautology [26]. The primary drawback of this technique is that it is limited to only detecting and preventing tautologies, which is only one of the many kinds of SQLIAs that our technique addresses.

Huang and colleagues also define a white-box approach for detecting input validation related errors that uses developer-provided annotations [13]. Relying on developer-provided annotations limits the practical applicability of the approach and the technique assumes that preconditions for all sensitive functions can be accurately expressed ahead of time, which is not always the case. Our technique is fully automated and does not require any developer intervention, such as annotations, in order to protect the application.

Recent work by Livshits and Lam [15] uses static analysis techniques to detect vulnerabilities that have been described using the PQL language [17]. In this approach, vulnerability signatures are described using PQL, and a static analyzer is generated from the vulnerability description. The analyzer detects instances of the vulnerability in the code. As opposed to our technique, this approach attempts to find known SQLIAs vulnerabilities in code as opposed to preventing them dynamically. Therefore, the approach can be effective in improving the code base of an application by identifying vulnerabilities in the program that can then be eliminated. However, the approach is limited, in that it can only detect known and specified vulnerabilities.

3. SQL INJECTION ATTACKS

Before describing our approach, we introduce a simple example of an SQL Injection Attack (SQLIA). This will be used as a running example throughout the paper as we describe our technique. We provide a more rigorous definition of SQLIAs in Section 3.2.

3.1 Example of SQL-Injection Attack

In this section we introduce an example of a web application that is vulnerable to an SQLIA and explain how an attacker could exploit this vulnerability. This particular example illustrates an attack based on injecting a tautology into the query string. Although tautologies represent only a subset of the SQLIAs that our technique can address, we use this type of attack for illustration because it is straightforward to understand and does not require deep knowledge of SQL syntax and semantics.

Figure 1 shows a typical web application in which a user on a client machine can access services provided by an application server and an underlying database. When the user enters a login and a password in the web form and presses the `submit` button, a URL is generated (`http://foo.com/show.jsp?login=doe&pass=xyz`) and sent to the web server. The figure illustrates which components of the web application handle the different parts of the URL.

In the example, the user input is interpreted by servlet `show.jsp`. (Servlets are Java applications that operate in conjunction with a Web server.) In this scenario the servlet would (1) use the user input to build a dynamic SQL query, (2) submit the query to the database, and (3) use the response from the database to generate HTML-pages that are then sent back to the user. Figure 2 shows an excerpt of a possible implementation of servlet `show.jsp`. Method `getUserInfo` is called with the login and the password provided by the user. If both `login` and `password` are empty, the method submits the following query to the database:

```
SELECT info FROM users WHERE login='guest'
```

If `login` and `password` are defined by the user, the method embeds the submitted credentials in the query. Therefore, if a user submits `login` and `password` as "`doe`" and "`xyz`," the servlet dynamically builds the query:

```
SELECT info FROM users WHERE login='doe' AND pass='xyz'
```

A web site that uses this servlet would be vulnerable to SQLIAs. For example, if a user enters "`' OR 1=1 --`" and "", instead of "`doe`" and "`xyz`", the resulting query is:

```
SELECT info FROM users WHERE login='' OR 1=1 --' AND pass=''
```

The database interprets everything after the WHERE token as a conditional statement, and the inclusion of the "`OR 1=1`" clause turns this conditional into a tautology. (The characters "`--`" mark the beginning of a comment, so everything after them is ignored.) As a result, the database would return information about all users. An attacker could insert a wide range of SQL commands via this exploit.

3.2 Definition of SQL-Injection Attack

An *SQL Injection Attack (SQLIA)* occurs when an attacker attempts to change the logic, semantics or syntax of a legitimate SQL statement by inserting new SQL keywords or operators into the statement.

This broad definition includes all of the variants of SQLIA reported in Anley's extensive documentation [1]. In particular, the definition includes, but is not limited to, attacks based on tautologies, injected additional statements, exploiting untyped parame-

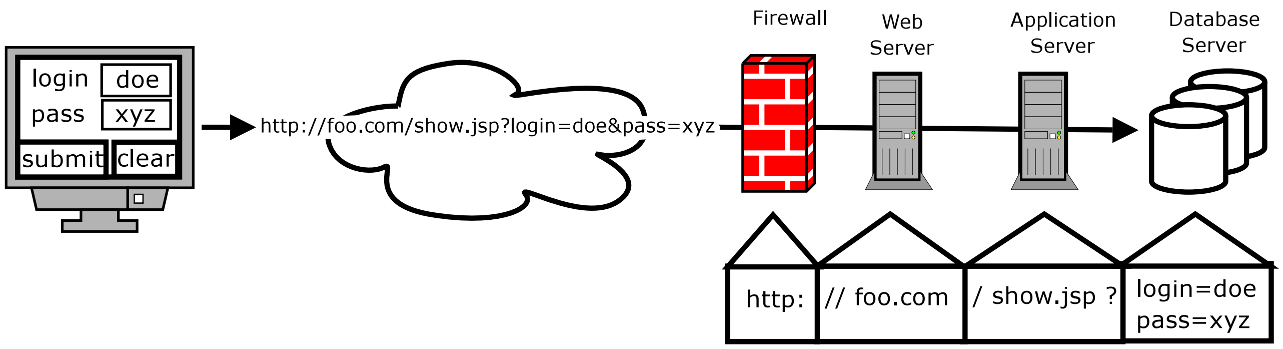


Figure 1: Example of interaction between a user and a typical web application.

ters, stored procedures, overly descriptive error messages, alternate encodings, length limits, second-order injections and injection of ”UNION SELECT”, ”ORDER BY”, and ”HAVING” clauses. (See [1] for a detailed explanations of the different types and forms of SQLIA.)

4. PROPOSED SOLUTION

Our proposed solution is a general technique that addresses all types of SQLIAs as defined in Section 3.2. The approach works by combining static analysis and runtime monitoring. The key intuition behind the approach is that (1) the source code contains enough information to infer models of the expected, legitimate SQL queries generated by the application, and (2) an SQLIA, by injecting additional SQL statements into a query, would violate such a model. In its static part, our technique uses program analysis to automatically build a model of the legitimate queries that could be generated by the application. In its dynamic part, our technique monitors the dynamically generated queries at runtime and checks them for compliance with the statically-generated model. Queries that violate the model represent potential SQLIAs and are thus prevented from executing on the database and reported.

The technique consists of four main steps. We summarize the steps and then describe them in more detail in subsequent sections.

Identify hotspots: Scan the application code to identify *hotspots*—points in the application code that issue SQL queries to the underlying database.

Build SQL-query models: For each hotspot, build a model that represents all the possible SQL queries that may be generated at that hotspot. A *SQL-query model* is a non-deterministic finite-state automaton in which the transition labels consist of SQL tokens (SQL keywords and operators), delimiters, and place holders for string values.

Instrument Application: At each hotspot in the application, add calls to the runtime monitor.

Runtime monitoring: At runtime, check the dynamically-generated queries against the SQL-query model and reject and report queries that violate the model.

4.1 Identify Hotspots

This step performs a simple scanning of the application code to identify hotspots. For the example servlet in Figure 2, the set of hotspots would contain a single element: the statement at line 10. (In Java-based applications, interactions with the database occur through calls to specific methods in the JDBC library,[2] such as `java.sql.Statement.execute(String)`.)

[2] `http://java.sun.com/products/jdbc/`

```
public class Show extends HttpServlet {
    ...
 1. public ResultSet getUserInfo(String login,
                                 String password) {
 2.   Connection conn = DriverManager.getConnection("MyDB");
 3.   Statement stmt = conn.createStatement();
 4.   String queryString = "";

 5.   queryString = "SELECT info FROM userTable WHERE ";
 6.   if ((! login.equals("")) && (! password.equals(""))) {
 7.     queryString += "login='" + login +
                       "' AND pass='" + password + "'";
      }
 8.   else {
 9.     queryString+="login='guest'";
      }
10.   ResultSet tempSet = stmt.execute(queryString);
11.   return tempSet;
    }
    ...
}
```

Figure 2: Example servlet.

4.2 Build SQL-Query Models

In this step we build the SQL-query model for each hotspot identified in the previous step. Within each hotspot, we are interested in computing the possible values of the query string passed to the database. To do this, we use the Java String Analysis (JSA) [5] library. This technique constructs a flow graph that abstracts away the control flow of the program and represents string-manipulation operations performed on string variables. For each string of interest the technique analyzes the flow graph and simulates the string-manipulation operations that are performed on the string. The result of the analysis is a Non-Deterministic Finite Automaton (NDFA) that expresses, at the character level, all the possible values the considered string can assume. The string analysis is conservative, so the NDFA for a string is an overestimate of all the possible values of the string.

It is worth noting that the JSA [5] technique generates Deterministic Finite Automata (DFAs), obtained by transforming each NDFA into a corresponding DFA. However, the transformation to DFAs increases the number of states and transitions in the graph and introduces cycles that complicate the construction of our SQL-query model. Therefore, we use their technique but skip its last step.

To build our SQL-query model for a given hotspot, we use the following process. We perform a depth first traversal of the NDFA for the hotspot and group characters as either SQL keywords, operators, or literal values and create a transition in the SQL-query model that is annotated with their literal value. For example, a sequence of transitions labeled 'S', 'E', 'L', 'E', 'C', and 'T' would be recognized as the SQL `SELECT` keyword and suitably grouped into a single transition labeled "SELECT". Because there are several SQL dialects each with their own set of keywords and operators, this part of the technique can be customized to recognize different dialects. We represent *variable strings* (i.e., strings that correspond to a variable related to some user input) using the symbol β (e.g., in our example the value of the variable `password` is represented as β.). This process is analogous to the one used by Gould, Su, and Devanbu [8], except that we perform it on NDFAs instead of DFAs.

Figure 3 shows the SQL-query model for the single hotspot in our example. The model reflects the two different query strings that can be generated by the code depending on the branch followed after the `if` statement at line 6 (Figure 2).

4.3 Instrument Application

In this step, we instrument the application by adding calls to the monitor that check the queries at runtime. For each hotspot, the technique inserts a call to the monitor before the call to the database. The monitor is invoked with two parameters: the string that contains the actual query about to be submitted and a unique identifier for the hotspot. Using the unique identifier, the runtime monitor is able to correlate the hotspot with the specific SQL-query model that was statically generated for that point and check the query against the correct model.

Figure 4 shows how the example application would be instrumented by our technique. The hotspot, originally at line 10 in Figure 2, is now guarded by a call to the monitor at line 10a.

```
...
10a. if (monitor.accepts (<hotspot ID>,
                          queryString))
     {
10b.     ResultSet tempSet =
                   stmt.execute(queryString);
11.      return tempSet;
     }
...
```

Figure 3: Example hotspot after instrumentation.

4.4 Runtime Monitoring

At runtime, the application executes normally until it reaches a hotspot. At this point, the string that is about to be submitted as a query is sent to the runtime monitor. The monitor parses the query string into a sequence of tokens according to the specific SQL syntax considered. Tokens in the query that represent string or numeric constants can match any transition in the SQL-query model labeled with β. Note that, in our parsing of the query string, the parser identifies empty string constants by their syntactic position and we denote these in the parsed query string using ε, the common symbol for the empty string. Figure 5 shows how the last two queries discussed in Section 3.1 would be parsed during runtime monitoring.

It is important to point out that our technique parses the query string the same way that the database would, according to the specific SQL grammar considered. In other words, it does not do a simple keyword matching over the query string, which would cause false positives and problems with user input that happened to match SQL keywords. For example, a user-submitted string that contains SQL keywords but is syntactically a text field, would be correctly recognized as a text field. However, if the user were to inject special characters, as in our example, to force part of the text to be evaluated as a keyword, the parser would correctly interpret this input as a keyword. Using the same parser as the database is important because it guarantees that we are interpreting the query the same way that the database will.

After the query has been parsed, the runtime monitor checks it by assessing whether the query violates the SQL-query model associated with the hotspot from which the monitor has been called. An SQL-query model is an NDFA whose alphabet consists of SQL keywords, operators, literal values, and delimiters, plus the special symbol β. Therefore, to check whether a query is compliant with the model, the runtime monitor can simply check whether the model (i.e., the automaton) accepts the query (i.e., whether a series of valid transitions reaches an accepting state). Recall that a string constant (including ε) or numeric constant in the parsed query string can match either β or an identical literal value in the SQL-query model.

If the model accepts the query, then the monitor lets the execution of the query continue. Otherwise, the monitor identifies the query as an SQLIA. In this case, the monitor prevents the query from executing on the database and reports the attack.

To illustrate, consider again the queries from Section 3.1 shown in Figure 5 and recall that the first query is legitimate, whereas the second one corresponds to an SQLIA. When checking query (a), the analysis would start matching from token $\boxed{\texttt{SELECT}}$ and from the initial state of the SQL-query model in Figure 3. Because the token matches the label of the only transition from the initial state, the automaton reaches the second state. Again, token $\boxed{\texttt{info}}$ matches the only transition from the current state, so the automaton reaches the third state. The automaton continues to reach new states until it reaches the state whose two outgoing transitions are labeled "`=`". At this point, the automaton would proceed along both transitions. On the upper branch, the query is not accepted because the automaton does not reach an accept state. Conversely, on the lower branch, all the tokens in the query are matched with labels on transitions, and the automaton reaches the accept state after consuming the last token in the query ("`'`"). The monitor can therefore conclude that this query is legitimate.

The checking of query (b) proceeds in an analogous way until token $\boxed{\texttt{OR}}$ in the query is reached. Because the token does not match the label of the only outgoing transition from the current state (`AND`), the query is not accepted by the automaton, and the monitor identifies the query as an SQLIA.

Once an SQLIA has been detected, our technique stops the query before it is executed on the database and reports relevant information about the attack in a way that can be leveraged by developers. In our implementation of the technique for Java, we throw an exception when the attack is detected and encode information about the attack in the exception. If developers want to access the information at runtime, they can simply leverage the exception-handling mechanism of the language and integrate their handling code into the application.

Having this attack information available at runtime is useful because it allows developers to react to an attack right after it is detected and develop an appropriate customized response. For example, developers may decide to avoid any risk and shut-down the part of the application involved in the attack. Alternatively, a developer could handle the attack by converting the information into a format that is usable by another tool, such as an Intrusion Detection System, and reporting it to that tool. Because this mechanism integrates with the application's language, it allows developers a good deal of flexibility in choosing a response to SQLIAs.

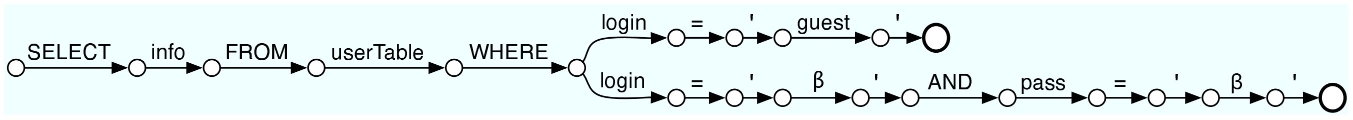
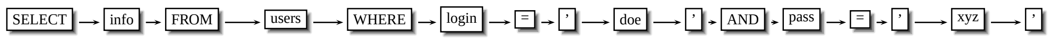


Figure 4: SQL-query model for the servlet in Figure 2.

```
(a) SELECT info FROM users WHERE login='doe' AND pass='xyz'
```



```
(b) SELECT info FROM users WHERE login='' OR 1=1 -- 'AND pass=''
```

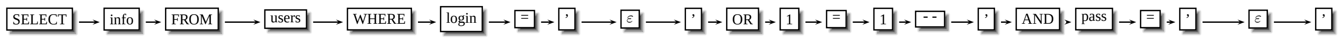


Figure 5: Example of parsed runtime queries.

Currently, the information reported by our technique includes the time of the attack, the location of the hotspot that was exploited, the attempted-attack query, and the part of the query that was not matched against the model. We are currently considering additional information that could be useful for the developer (e.g., information correlating program execution paths with specific parts of the query model) and investigating ways in which we can modify the static analysis to collect this information.

5. ADDITIONAL CONSIDERATIONS

5.1 Efficiency

In terms of efficiency, our approach requires the execution of the monitor for each database query. Since we check the query against the SQL-query model, which is an NDFA, the worst case complexity is exponential in the size of the automaton, which in the worst case is quadratic in the size of the program [5]. However, the visit of the NDFA is typically linear because the automata generated by the analysis are usually trees. Also, the automata are linear for typical programs—the case of a quadratic size corresponds to an application that modifies the query string and branches in each line of the program. In fact, our experience is that most automata are actually quite small with respect to the size of the corresponding application. In practice, considered that the monitor just checks a typically short set of tokens against an NDFA, whereas database queries normally involve interactions over a network, we expect the overhead for the monitoring to be negligible. This intuition is confirmed by our empirical evaluation (see Section 6).

5.2 Effectiveness and Precision

Our technique can produce false negatives in two situations: (1) when the string analysis results in a SQL query model that is overly conservative and includes spurious queries (i.e. queries that could not be generated by the application) that happen to match an attack; and (2) when a legitimate query happens to have the same "SQL structure" of an attack. For example, if a developer adds conditions to a query from within a loop, an attacker who inserts an additional condition of the same type would generate a query that does not violate the SQL-query model. We expect both of these cases to be rare in practice because of the typically peculiar structure of SQLIAs. The attacker would have to produce an attack that directly matches either an imprecision of the analysis or a specific pattern. Moreover, in both cases, the type of attacks that could be exploited would be limited by the constraints imposed by the rest of the model that was used to match the query.

Although the string analysis that we use is conservative, there are situations in which our technique can produce false positives. False positives can occur when the string analysis is unable to generate a string model that is precise enough. In particular, if the string analysis over-approximates a hard-coded string and this hard-coded string is used in the application to construct a SQL token, our technique will generate an incomplete SQL-query model. The problem is that, in these cases, our analysis cannot determine whether the over-approximation represents a variable or a hard-coded SQL token. Our analysis could either allow queries to partially match keywords and variables (possibly causing false negatives) or reject queries that traverse those particular transitions (possibly causing false positives). We chose a conservative approach and decided to reject the queries in these cases. It is worth noting that this situation did not occur in any of the 271 automata that we generated for our evaluation.

Lastly, it is important to note the scope of our technique. Our technique targets SQLIAs, which are exploits where an attacker is attempting to inject SQL statements into a query sent to the database. SQLIAs *do not* include other types of web-application-related attacks. For example, a common attack on web applications is to steal authentication tokens that are passed between a browser and the database application to hijack the session and issue queries to the database. This is not an injection attack because the attacker is simply using the rights and privileges assigned to the victim to access the database through legitimate (i.e., non injected) queries.

5.3 Assumptions

Our technique requires two assumptions to be satisfied by the targeted web application. The first assumption is that user input consists only of values, such as numbers and strings, that are not meant to be interpreted as SQL tokens. In other words, our technique cannot handle cases in which the user input is legitimately supposed to add SQL tokens to the query. Applications that allow the user to do so would cause our technique to generate false positives because we would recognize the user-introduced SQL tokens and operators as an injection. (While this is technically correct according to our definition of SQLIA, because the user input is an actual injection of SQL tokens, blocking these queries would result in incorrect behavior for this type of application.) In general, this is a reasonable assumption. Users can still enter SQL statements in a field (e.g., a text area of a discussion board on SQL), as long as the application treats such input as text. Applications that do allow users to directly enter SQL queries are typically database-administration tools whose usage is restricted to a trusted set of users connecting from specific machines.

Second, we assume that application developers build database queries by combining substrings and that they do not obfuscate the logic of the query construction. This assumption is important because our technique's safety and precision depends on the accuracy of the underlying string analysis, which in turn depends on the complexity of the query-generation code. Because of this assumption, our technique cannot handle applications that use different query-development paradigms, such as SQL DOM [18], and applications that store query strings in a way that prevents the string analysis from detecting them (e.g., in a file). We believe that this assumption is not too restrictive. In fact, it is satisfied by all of the web applications that we have analyzed so far. Also, the same assumption has been widely adopted by other researchers [1, 3, 5, 7, 8, 11, 12, 13, 16, 19, 20, 21, 22, 24, 26].

6. EVALUATION

The goal of our empirical evaluation is to assess the effectiveness and efficiency of the technique presented in this paper when applied to various web applications. We developed a prototype tool, called AMNESIA, that implements the technique, and used it to perform an empirical study on a set of subjects. The study investigates three research questions:

RQ1: What percentage of attacks can our technique detect and prevent that would otherwise go undetected and reach the database?

RQ2: How much overhead does our technique impose on web applications at runtime?

RQ3: What percentage of legitimate accesses does our technique identify as false positives?

The following sections present the tool, illustrate the setup for our evaluation, and discuss the two studies that we performed to address our research questions.

6.1 The Tool: AMNESIA

AMNESIA (Analysis and Monitoring for NEutralizing SQL Injection Attacks) is the prototype tool that implements our technique to counter SQLIAs for Java-based web applications. AMNESIA is developed in Java and its implementation consists of three modules that leverage various existing technologies and libraries:

Analysis module. This module implements Steps 1 and 2 of our technique. Its input is a Java web application and it outputs a list of hotspots and a SQL-query models for each hotspot. For the implementation of this module, we leveraged JSA [5]. The analysis module is able to analyze Java Servlets as well as JSP pages.

Instrumentation module. This module implements Step 3 of our technique. It inputs a Java web application and a list of hotspots and instruments each hotspot with a call to the runtime monitor. We implemented this module using INSECT, a generic instrumentation and monitoring framework for Java developed at Georgia Tech [4].

Runtime-monitoring module. This module implements Step 4 of our technique. It takes as input a query string and the ID of the hotspot that generated the query, retrieves the SQL-query model for that hotspot, and checks the query against the model. For this module, we also leveraged INSECT.

Figure 6 shows a high-level overview of AMNESIA. In the static phase, the Instrumentation Module and the Analysis Module take as input a web application and produce (1) an instrumented version of the application, and (2) an SQL-query model for each hotspot in the application. In the dynamic phase, the Runtime-Monitoring Module checks the dynamic queries while users interact with the web application. If a query is identified as an attack, it is blocked and reported.

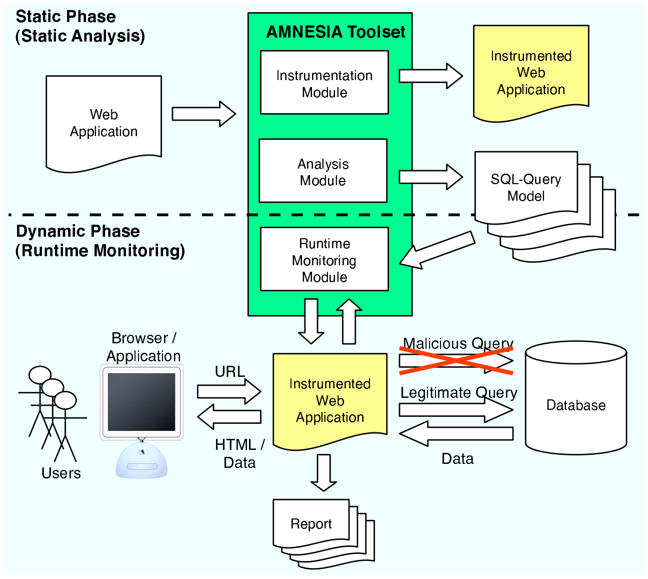


Figure 6: High-level overview of AMNESIA.

6.2 Experiment Setup

To be able to investigate our research questions, we needed a test bed for our technique and tool. In particular, we needed a set of web applications and a set of inputs for those applications that included both legitimate inputs and SQLIAs. In the next two sections, we present the subjects that we collected for our studies and describe how we generated a large, meaningful set of inputs for the subjects. In our experimental setup, we were careful to develop the test bed in a way that allows us (and other researchers) to easily rerun our experiments.

6.2.1 Subjects

For the evaluation we used seven experimental subjects. The subjects are all typical web applications that accept input from the user through a web form and use the input to build queries to an underlying database. Five of the subjects, Employee Directory, Bookstore, Events, Classifieds, and Portal, are *commercial* applications that we obtained from GotoCode (`http://www.gotocode.com`). The two remaining applications, Checkers and Office Talk, were developed by different teams of students as part of a class project. Although these last two are not commercial applications, we selected them because they have been used in previous, related studies [8]. All subjects were deployed on our local test bed servers as they would be in a commercial setting.

Table 1 provides information about the subjects. For each subject, the table shows its name (*Subject*); a concise description (*Description*); its size in terms of lines of code (*LOC*); the number of accessible servlets (*Servlets*) with the total number of servlets in the application in parenthesis; the number of injectable parameters (*Injectable Params*); the number of state parameters (*State Params*); the number of hotspots (*Hotspots*); and the average size of the SQL automata generated by AMNESIA (*Automata Size*), with the minimum–maximum range in parentheses.

We define an *injectable parameter* as an input parameter to a web application whose value is used to build part of a query that is

Subject	*Description*	*LOC*	*Servlets*	*Injectable* Params	*State* Params	*Hotspots*	*Automata Size* Size (#nodes)
Checkers	Online checkers game	5,421	18 (61)	44	0	5	289 (2–772)
Office Talk	Purchase-order management system	4,543	7 (64)	13	1	40	40 (8–167)
Employee Directory	Online employee directory	5,658	7 (10)	25	9	23	107 (2–952)
Bookstore	Online bookstore	16,959	8 (28)	36	6	71	159 (2–5,269)
Events	Event tracking system	7,242	7 (13)	36	10	31	77 (2–550)
Classifieds	Online management system for classifieds	10,949	6 (14)	18	8	34	91 (2–799)
Portal	Portal for a club	16,453	3 (28)	39	7	67	117 (2–1,187)

Table 1: Subject programs for the empirical study.

then sent to the database. We define a *state parameter* as a parameter that may affect the flow of control within the web application but never becomes part of a query. By definition, state parameters cannot result in SQL injection, and therefore, we only focus on injectable parameters for our attacks. For example, in URL `http://some.host/empldir/Login.jsp?Login=john&Password=xyz&FormAction=login`, for application Employee Directory, *Login* and *Password* are injectable parameters, whereas *FormAction* is a state parameter that the web application uses to decide how to handle the user request.

We did not consider all servlets in our study because some servlets can only be accessed if the web session is in a very specific state, which is difficult to recreate automatically and would require custom handling of each web application. We call *accessible servlets* the servlets that, to be accessed, only required the user to be logged-in or did not require sessions at all. Because we were able to generate enough attacks considering accessible servlets only, we did not consider the remaining servlets.

6.2.2 Input Generation

For our evaluation we generated a large set of inputs that represented normal and malicious usage of the subject applications. The attacks were developed by a Masters student at Georgia Tech who worked for a local software-security company. The student was an experienced programmer who had developed commercial-level penetration tools for detecting SQL-injection vulnerabilities. Moreover, the student was not familiar with our technique, which reduced the risk of developing a set of attacks biased by the knowledge of the approach and its capabilities. In the rest of this section, we outline the specific steps that were taken in order to generate the set of inputs and prepare them for usage in our evaluation.

First, we identified all the servlets in each subject. Each servlet is a URL that accepts a set of parameters and that is typically accessed via the submission of a web form. For example, in the login page of the example in Figure 1, the servlet is `http://foo.com/show.jsp`, and a generic access to the entry point is `http://foo.com/show.jsp?login=`⟨`value1`⟩`&pass=`⟨`value2`⟩. For the accessible servlets, we identified the corresponding injectable and state parameters. This step was necessary because if state parameters are not assigned the correct value, the web application simply returns an error, and no attack can be successful. Further, we classified the type of each parameter as either string or numeric and identified the specific values that could be used for the state parameters. (This part of the evaluation was performed manually and involved a considerable effort.) At the end of this first phase, we had two lists of parameters, state and injectable, for each application and servlet; each state parameter was associated with a set of possible values and each injectable parameter was classified as either string or numeric.

Second, the student generated a set of *attack strings*, malicious input strings that can be used to perform SQLIAs. To define the set of attack strings, the student used exploits that were developed by commercial penetration teams to take advantage of SQL-injection vulnerabilities. The student also surveyed various sources that included US-CERT (`http://www.us-cert.gov/`), CERT/CC Advisories (`http://www.cert.org/advisories/`), and several security-related mailing lists. The student used the examples of attacks discussed on those sites and mailing lists to define the attack strings. In this way, the student generated attack strings that are representative of real attacks that could be performed on the considered subjects. The set contained a total of 30 attack strings that encoded a wide variety of attacks. The attack strings themselves were each unique and represented a different way to exploit an SQLIA vulnerability. Most of the attack strings had been previously reported in literature [1]. These strings included attacks based on injections of additional statements, exploiting untyped parameters, stored procedures, tautologies, alternate encodings, and injections of ”UNION SELECT”, ”ORDER BY”, ”HAVING” and ”JOIN” clauses. Other types of attacks, such as the ones that leverage overly descriptive error messages and the ones based on second-order injections were not included since they depended on an initial first round of successful injection.

Third, the student generated two sets of inputs for each application. Each input consisted of the URL for a servlet together with the value(s) of one or more parameters of that servlet. The first set is the set of legitimate inputs, which we call LEGIT, and contains only inputs with legitimate parameters (i.e., inputs that cannot result in an SQLIA). The second set is the set of SQL-injection inputs, which we call ATTACK, and consists of inputs such that at least one of the parameters is an attack string (i.e., inputs that may result in an SQLIA). To populate set LEGIT, the student created different inputs by using different combinations of legitimate values for the injectable parameters based on their type. All state parameters were always assigned a meaningful and correct value. For each application, the LEGIT set contained 500 elements.

To populate the ATTACK sets, the student used two kinds of inputs: (1) inputs for which one parameter is assigned an attack string (and all other parameters are assigned legitimate values); (2) inputs for which more than one parameter is assigned an attack string. The student created inputs of the first type by exhaustively assigning to each parameter of each servlet all possible attack strings, one at a time. The remaining parameters were assigned various combinations of legitimate values based on their type. Then, all unsuccessful attacks were eliminated from the set, that is, all attacks that were blocked by the web application were removed from the ATTACK set. (Some of the web applications performed sufficient input validation and returned an error page when they detected improper input.) The student generated inputs of the second type for a web application by randomly selecting, without repetition, an accessible servlet within the application and a set of possible values for all the parameters. For each parameter, a random number between 1 and 60 was generated. If the number was less then or equal to 30, the student assigned the corresponding attack string to the parameter. Otherwise, he assigned a legitimate value to the parameter. In this way, each parameter had a 50% chance of containing an attack string. The student continued generating inputs of the second type until we had either 100 successful attacks (i.e., attacks that were not blocked by the web application and reached the database) or 2,000 unsuccessful attacks. (Note that the random generation was always able to generate at least 100 successful attacks, so we never reached the threshold of unsuccessful attacks.) For each application in the evaluation we had an ATTACK set whose size ranged from 140 to 280 elements. Table 2, explained in the next section, shows the number of attacks for each application. In the table, we also report the number of unsuccessful attacks for completeness.

6.3 Study 1: Effectiveness

In the first study, we investigated **RQ1**, the effectiveness of our technique in detecting and preventing SQLIAs. We analyzed and instrumented each application using AMNESIA and ran all of the inputs in each of the applications' ATTACK sets. For each application, we measured the percentage of attacks detected and reported by AMNESIA. (As previously discussed, when AMNESIA detects an attack, it throws an exception, which is in turn returned by the web application. Therefore, it is easy to accurately detect when an attack has been caught.)

The results for this study are shown in Table 2. The table shows for each subject the number of unsuccessful attacks (*Unsuccessful*), the number of successful attacks (*Successful*), and the number of attacks detected and reported by AMNESIA (*Detected*), both in absolute terms and as a percentage over the total number of successful attacks, in parentheses. As the table shows, AMNESIA achieved a perfect score. For all subjects, it was able to correctly identify all attacks as SQLIAs, that is, it generated no false negatives.

Subject	*Unsuccessful*	*Successful*	*Detected*
Checkers	1195	248	248 (100%)
Office Talk	598	160	160 (100%)
Employee Directory	413	280	280 (100%)
Bookstore	1028	182	182 (100%)
Events	875	260	260 (100%)
Classifieds	823	200	200 (100%)
Portal	880	140	140 (100%)

Table 2: Results of Study 1.

6.4 Study 2: Efficiency and Precision

In the second study, we investigated **RQ2** and **RQ3**. To investigate **RQ2** (i.e., the efficiency of our technique), we ran all of the legitimate inputs in the LEGIT sets on the original (i.e., not instrumented) web applications and measured the response time of the applications for each web request. We then ran the same inputs on the versions of the web applications instrumented by AMNESIA and again measured the response time. Finally, we computed the differences between the response times in the two cases, which corresponds to the overhead imposed by our technique.

We found that the overhead imposed by our technique is negligible and, in fact, barely measurable, ranging from 10 to 40 milliseconds. This amount of time should be considered an upper bound on the overhead, as our implementation was not written to be efficient. For example, we accessed file IO each time we checked a query, both to load the model and to log the results. In a more performance oriented implementation, these IO accesses would be minimized and cached to improve performance. The overhead results confirm our expectations. Intuitively, the time for the network access and the database transaction completely dominates the time required for the runtime checking. As the results show, our technique is efficient and can be used without affecting the response time of a web application in a meaningful manner.

To investigate **RQ3** (i.e., the rate of false positives generated by our technique), we simply assessed whether AMNESIA identified any legitimate query as an attack. The results of the assessment were that AMNESIA correctly identified all such queries as legitimate queries and reported no false positives.

6.5 Discussion

The results of our study are fairly clear cut. For all subjects, our technique was able to correctly identify all attacks as SQLIAs, while allowing all legitimate queries to be performed. In other words, for the cases considered, our technique generated no false positives and no false negatives. The lack of false positives and false negatives is very promising and provides evidence of the viability of the technique.

In our study, we did not compare our results with alternative approaches against SQLIAs because many of the automated approaches that we are aware of, only address a small subset of the possible SQLIAs. (For example, the approach in [8] is focused on type safety, and the one in [26] focuses only on tautologies.) Therefore, we know analytically that such approaches would not be able to identify many of the attacks in our test bed.

As for all empirical studies, there are some threats to the validity of our evaluation, mostly with respect to external validity. The results of our study may be related to the specific subjects considered and may not generalize to other web applications. To minimize this risk, we used a set of real web applications (except for the two applications developed by students teams) and an extensive set of realistic attacks. Although more experimentation is needed before drawing definitive conclusions on the effectiveness of the technique, the results we obtained so far are promising.

7. CONCLUSION AND FUTURE WORK

We presented a new fully automated technique for detecting, preventing, and reporting SQL Injection Attacks (SQLIAs). The technique is based on the intuition that the web-application code implicitly contains a "policy" that allows for distinguishing legitimate and malicious queries. To extract and check this policy, the technique combines static and dynamic analysis. In the static phase, the technique uses an existing string analysis to extract from the web-application's code a model of all the query strings that could be generated by the application. In the dynamic phase, the technique monitors dynamically-created queries for conformance with the statically-built model. Queries that are not compliant with the model are identified as SQLIAs, blocked, and relevant information is reported to developers and administrators.

In this paper, we also presented AMNESIA, a prototype tool that implements our technique for Java-based web applications, and an empirical evaluation of the technique. The empirical evaluation of the technique was performed on a set of seven web applications that consisted of two applications developed by a student team, but also used by other researchers, and five real applications. AMNESIA was able to stop all of the 1,470 attacks that we performed on the considered applications without producing any false positive for the 3,500 legitimate accesses to the applications. Furthermore, AMNESIA proved to be quite efficient in practice, at least for the cases considered—the overhead imposed by the technique on the web application was barely measurable.

In our future work we will investigate alternate techniques for building SQL models for cases in which the static analysis cannot be used (e.g., because of scalability issues). In particular, we will study possible alternative static analysis techniques. In our current approach, we use a very precise (and expensive) analysis [5] to build the character-level model that we then compact and transform into our SQL-query model. Our technique is mostly interested in SQL keywords and operators, and the strings representing them are typically constant strings that are rarely manipulated in the application. Therefore, we may be able to use a simplified string analysis to extract the SQL-query models that our monitoring technique needs directly from a suitable representation of the code. Finally, we will investigate combined static and dynamic approaches for building models to handle cases in which the static analysis can be successfully applied on only a subset of the application.

Acknowledgments

This work was supported in part by NSF awards CCR-0306372, CCR-0205422, and CCR-0209322. Wenke Lee and David Dagon provided useful comments on an early version of the paper. Carl Gould, Zhendong Su, and Premkumar Devanbu provided us with an implementation of their JDBC CHECKER tool and with the set of subjects they used in their experimentation. Gary Wassermann provided useful feedback and comments on our drafts. Jeremy Viegas helped to develop our test bed infrastructure and devoted long hours to the generation of the input data for our experiment.

8. REFERENCES

[1] C. Anley Advanced SQL Injection In SQL Server Applications. Next Generation Security Software Ltd. White Paper, 2002.

[2] D. Aucsmith. Creating and maintaining software that resists malicious attack. `http://www.gtisc.gatech.edu/aucsmith_bio.htm` Distinguished Lecture Series. Atlanta, GA. September 2004.

[3] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL injection attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, pages 292–302, June 2004.

[4] A. Chawla and A. Orso. A generic instrumentation framework for collecting dynamic information. In *Online Proceeding of the ISSTA Workshop on Empirical Research in Software Testing (WERST 2004)*, Boston, MA, USA, July 2004. `http://www.sce.carleton.ca/squall/WERST2004`.

[5] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise Analysis of String Expressions. In *Proceedings of the 10th International Static Analysis Symposium, SAS 03*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003.

[6] W. R. 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 2005)*, 2005.

[7] 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 2004) – Formal Demos*, pages 697–698, 2004.

[8] C. Gould, Z. Su, and P. Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 645–654, 2004.

[9] A. R. Group. Java Architecture for Bytecode Analysis (JABA), 2004. `http://www.cc.gatech.edu/aristotle/Tools/jaba.html`.

[10] W. G. Halfond and A. Orso. Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks. In *Proceeding of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005)*, St. Louis, MO, USA. May 2005.

[11] M. Howard and D. LeBlanc. *Writing Secure Code*. Microsoft Press, Redmond, Washington, 2nd edition, 2003.

[12] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In *Proceedings of the 11th International World Wide Web Conference (WWW 2003)*, May 2003.

[13] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. T. Lee, and S.-Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the 12th International World Wide Web Conference (WWW 2004)*, May 2004.

[14] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks with Instruction-Set Randomization. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS 2003)*, pages 272–280, October 2003.

[15] V. B. Livshits and M. S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis In *Usenix Security Symposium*, Aug. 2005.

[16] O. Maor and A. Shulman. SQL Injection Signatures Evasion. `http://www.imperva.com/application_defense_center/white_papers/sql_injection_signatures_evasion.html`, April 2004. White paper.

[17] M. Martin, V. B. Livshits, and M. S. Lam. Finding application errors using PQL: a program query language. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA), Oct. 2005.

[18] R. McClure and I. Krüger. SQL DOM: Compile Time Checking of Dynamic SQL Statements. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 88–96, 2005.

[19] S. McDonald. SQL Injection: Modes of attack, defense, and why it matters. `http://www.governmentsecurity.org/articles/SQLInjectionModesofAttackDefenceandWhyItMatters.php`, April 2004. White paper.

[20] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, David Evans. Automatically Hardening Web Applications Using Precise Tainting Information In *Twentieth IFIP International Information Security Conference (SEC 2005)*, May 2005.

[21] OWASPD – Open Web Application Security Project. Top ten most critical web application vulnerabilities. `http://www.owasp.org/documentation/topten.html`, 2005.

[22] T. Pietraszek1 and C. V. Berghe. Defending Against Injection Attacks through Context-Sensitive String Evaluation. In *Proceedings of Recent Advances in Intrusion Detection (RAID2005)*, 2005.

[23] D. Scott and R. Sharp. Abstracting Application-level Web Security. In *Proceedings of the 11th International Conference on the World Wide Web (WWW 2002)*, pages 396–407, 2002.

[24] SecuriTeam. SQL Injection Walkthrough. `http://www.securiteam.com/securityreviews/5DP0N1P76E.html`, May 2002. White paper.

[25] F. Valeur and 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), July 2005.

[26] G. Wassermann and Z. Su. An Analysis Framework for Security in Web Applications. In *Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2004)*, pages 70–78, 2004.