Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks

William G.J. Halfond, Alessandro Orso, and Panagiotis Manolios
College of Computing – Georgia Institute of Technology
{whalfond, orso, manolios}@cc.gatech.edu

ABSTRACT
SQL injection attacks pose a serious threat to the security of Web applications because they can give attackers unrestricted access to databases that may contain sensitive information. In this paper, we propose a new, highly automated approach against SQL injection that has both conceptual and practical advantages over most existing approaches. From the conceptual standpoint, the approach is based on the novel idea of positive tainting and the concept of syntax-aware evaluation. From the practical standpoint, our technique is at the same time precise and efficient and has minimal deployment requirements. The paper also describes WASP, a tool that implements our technique, and a set of studies performed to evaluate our approach. In the studies, we used our tool to protect several Web applications and then subjected them to a large and varied set of attacks and legitimate accesses. The evaluation was a complete success: WASP successfully and efficiently stopped all of the attacks without generating any false positives. Both our tool and experimental infrastructure are available to other researchers and practitioners.

1. INTRODUCTION
SQL injection attacks (SQLIAs) have been described as one of the most significant threats to Web-application security [5]. A vulnerability to SQL injection can give attackers complete access to the potentially sensitive or confidential information contained in the databases underlying an application. In spite of the significant impact of SQL injection attacks, many Web applications remain vulnerable to such attacks.

The root cause of SQL injection vulnerabilities is relatively simple: attackers take advantage of an application’s lack of input validation to introduce attack strings that contain specially-encoded database commands. When these strings are used by an application to build a query, the attacker’s embedded commands are then interpreted and executed by the database. Although this mechanism is well understood, prevention using good defensive coding practices has not been completely successful in eliminating the problem. The reason for this is that it is difficult to implement and enforce a rigorous defensive coding discipline. Moreover, defensive coding is problematic in the case of legacy software because of the cost and complexity of retrofitting existing code. Researchers have also proposed a wide range of alternative techniques to address SQLIAs, but many of these have limitations that affect their effectiveness and practicality.

In this paper we propose a new, highly automated approach for detecting and preventing SQLIAs. Intuitively, our approach works by (1) identifying “trusted” strings in an application, and (2) allowing only these trusted strings to be used to create sensitive parts of SQL query strings, such as keywords and operators. The general mechanism that we use to implement this approach is based on the idea of dynamic taint propagation—input is marked and tracked during usage by the application and prevented from being used in ways that could cause harm to the system.

While numerous techniques have been proposed in the literature for addressing the SQLIA problem, our use of dynamic positive tainting has several important advantages, which we now outline. Techniques based on computationally expensive or complex static analyses can introduce imprecision in the results and do not scale well (e.g., [9, 13, 14]). In contrast, because positive tainting is purely dynamic, it is both precise and efficient. Many previously proposed techniques require extensive involvement by the developer. For example, developers often need to manually rewrite parts of their applications, build queries using special libraries, insert calls to special checking functions, or mark all points in the code at which malicious input could be introduced (e.g., [3, 4, 12, 14, 17, 22]). In contrast, our approach is highly automated and, in most cases, requires minimal or no developer intervention. Lastly, several techniques involve the deployment of extensive infrastructure and require complex configurations (e.g., [2, 21, 23]). Our approach does not require the installation of any additional infrastructure and can be deployed automatically.

While there has been recent work in using dynamic tainting to address the problem of SQL injection (e.g., [8, 19, 20]), our approach makes several conceptual and practical improvements that take advantage of the specific characteristics of SQLIAs.

The conceptual advantages of our approach are in the use of positive tainting and a flexible syntax-aware evaluation. Positive tainting focuses on the identification and marking of trusted data. In contrast, the standard “negative” tainting identifies and marks untrusted data. In the context of preventing SQLIAs, there are several reasons why positive tainting is more effective than negative tainting. First, in Web applications, trusted data sources can be more readily identified than untrusted data sources; therefore, the use of positive tainting leads to increased automation. Second, and more importantly, the two approaches differ significantly with respect to incompleteness. When using positive tainting, failure to identify the set of trusted data sources can only lead to false positives, and such incompleteness tends to be easy to detect and correct during testing. In contrast, when using approaches based on negative tainting, failure to identify all untrusted data sources will leave the application vulnerable to attacks. In addition, such vulnerabilities are difficult to detect during testing, in part because untrusted data may come from sources such as user input, uploaded files, browser cookies, and local server variables. Failure to identify one of these sources as untrusted can result in SQLIAs in the field that may never be discovered; in contrast, incompleteness with positive tainting will result in false positives, which are undesirable, but whose presence can be detected immediately and which can be easily corrected.
The second conceptual advantage of our approach is due to its use of flexible syntax-aware evaluation, which provides developers with a mechanism to regulate the usage of strings based not only on their source, but also on the syntactical role they play in the final query string. This allows developers to use a wide range of external sources of input to build queries, while protecting them from possible attacks introduced via these sources.

From a practical standpoint, the advantages of our approach are that it is accurate, efficient, and has only minimal deployment requirements. For taint propagation and marking, we use a library called MetaStrings. MetaStrings tracks taint information at the character level and accurately maintains markings during string operations. Our approach efficiently uses instrumentation to fully automate the usage of the MetaStrings library in the Web application and adds calls to perform the syntax-aware evaluation. Finally, our approach is completely defined at the application level and can be used without installing specialized runtime systems.

In the rest of the paper, we present our approach and its technical components. We also discuss the results of an extensive empirical evaluation of the effectiveness and efficiency of our technique. For this evaluation, we have implemented our approach in a tool called WASP (Web Application SQL-injection Preventer). We evaluated WASP on a set of seven Web applications of various types and sizes. For each application, we protected it with WASP and then targeted it with a large and varied set of attacks and legitimate accesses. This assessed the ability of our technique to detect and prevent attacks without stopping legitimate accesses to the database. The results of the evaluation show that our technique was able to stop all of the attacks without generating false positives for any of the legitimate accesses. Moreover, our technique proved to be very efficient and imposed little overhead on the Web applications.

The main contributions of this work are:

- A new, automated technique for preventing SQL-injection attacks based on the novel concept of positive tainting and flexible syntax-aware evaluation.
- A mechanism for precisely and efficiently assigning trust markings to strings in Java programs and maintaining the markings at runtime, while strings are manipulated.
- A tool that implements the technique for Java-based Web applications and imposes minimal deployment requirements.
- An empirical evaluation of the technique that shows its effectiveness and efficiency.

The rest of this paper is organized as follows. Section 2 provides an example of an SQL injection attack. We discuss our approach in Section 3 and the prototype implementation of our technique in Section 4. Section 5 presents our empirical evaluation. In Section 7, we review and discuss related work. Finally, we conclude and outline future work in Section 7.

2. SQL INJECTION ATTACKS

Su and Wassermann provide a formal definition of SQL Injection Attack (SQLIA) in [22]. Intuitively, an SQLIA occurs when an attacker changes the developer’s intended structure of an SQL command by inserting new SQL keywords or operators. SQLIAs leverage a wide range of mechanisms and input channels to inject malicious commands into a vulnerable application [10]. In this section, we introduce an example application that contains a SQL injection vulnerability and show how an attacker can leverage the vulnerability to perform an SQLIA. Note that the example represents an extremely simple form of attack, and we present it for illustrative purposes only. Interested readers may refer to References [1] and [10] for further examples of the different types of SQLIAs.

The code excerpt in Figure 1 represents an implementation of a login functionality that we could find in a typical Web application. This type of login function would commonly be part of a Java servlet, a type of Java application that runs on a Web application server, and whose execution is triggered by the submission of a URL from a user of the Web application. The servlet in the example uses the input parameters login and pin to dynamically build an SQL query or command. The login and pin are checked against the credentials stored in the database. If they match, the corresponding user’s account information is returned. Otherwise, a null set is returned by the database and the authentication fails. The servlet then uses the response from the database to generate HTML pages that are sent back to the user’s browser by the Web server. Given the servlet code, if a user submits login and pin as “doe” and “123,” the application dynamically builds the query:

```
SELECT acct FROM users WHERE login='doe' AND pin=123
```

If the login and password match the corresponding entry in the database, doe’s account information is returned and then displayed by function displayAccount(). If there is no match in the database, function sendAuthFailed() displays an appropriate error message. An application that uses this servlet is vulnerable to SQLIAs. For example, if an attacker enters “admin” ‘ --’ as her user name instead of “doe”, and input any value for the pin (e.g., “0”), the resulting query is:

```
SELECT acct FROM users WHERE login='admin' --' AND pin=0
```

This query allows an attacker to authenticate herself as the administrator. This attack succeeds because, in SQL, “--” is the comment operator, and everything after it is ignored. Therefore, the database simply searches for an entry where login is equal to admin and returns that database record. After the “successful” login, the function displayAccount() would reveal the admin’s account information to the attacker.

1. String login = getParameter("login");
2. String pin = getParameter("pin");
3. Statement stmt = connection.createStatement();
4. String query = "SELECT acct FROM users WHERE login='" + login + "' AND pin=" + pin;
5. query += login + "' AND pin=" + pin;
6. ResultSet result = stmt.executeQuery(query);
7. if (result != null)
8. displayAccount(result); // Show account
9. else
10. sendAuthFailed(); // Authentication failed

Figure 1: Excerpt of a Java servlet implementation.

The main contributions of this work are:

- A new, automated technique for preventing SQL-injection attacks based on the novel concept of positive tainting and flexible syntax-aware evaluation.
- A mechanism for precisely and efficiently assigning trust markings to strings in Java programs and maintaining the markings at runtime, while strings are manipulated.
- A tool that implements the technique for Java-based Web applications and imposes minimal deployment requirements.
- An empirical evaluation of the technique that shows its effectiveness and efficiency.

The results of the evaluation show that our technique was able to stop all of the attacks without generating false positives for any of the legitimate accesses. Moreover, our technique proved to be very efficient and imposed little overhead on the Web applications. The main contributions of this work are:

- A new, automated technique for preventing SQL-injection attacks based on the novel concept of positive tainting and flexible syntax-aware evaluation.
- A mechanism for precisely and efficiently assigning trust markings to strings in Java programs and maintaining the markings at runtime, while strings are manipulated.
- A tool that implements the technique for Java-based Web applications and imposes minimal deployment requirements.
- An empirical evaluation of the technique that shows its effectiveness and efficiency.

The rest of this paper is organized as follows. Section 2 provides an example of an SQL injection attack. We discuss our approach in Section 3 and the prototype implementation of our technique in Section 4. Section 5 presents our empirical evaluation. In Section 7, we review and discuss related work. Finally, we conclude and outline future work in Section 7.

3. OUR APPROACH

Our approach is based on dynamic tainting, which has been widely used to address security problems related to input validation. The general approach of dynamic tainting is to mark as tainted certain data (typically, user input) and track the flow of such data throughout a program. In general, tainted data is restricted from being used in ways that could potentially be harmful to the system. Our technique, while using dynamic tainting, differs from the general approach by making several conceptual and practical improvements. We first overview these differences and then discuss them in detail in the next sections. First of all, unlike all existing techniques, we base our approach on the novel concept of positive tainting—the identification and marking of trusted data (strings, in our case) instead of untrusted data. Second, we perform an accurate taint prop-

1 For simplicity, in the rest of this paper we use the terms query and command interchangeably.
Stopped attacks

Untrusted data

Identified trusted data

Identified untrusted data

Trusted data

Successful attacks

False positives

Figure 2: Identification of trusted and untrusted data.

Agitation. We accurately propagate these trust markings by tracking them at the character level and by leveraging object encapsulation to precisely account for the semantics of string operations. Third, at each point in the application where a query is executed, we perform a syntax-aware evaluation of the query string before it is sent to the database. We check that all non-literal parts of the query (i.e., all SQL keywords and operators) consist of only trusted characters. If the check fails, the query is blocked from executing on the database. Fourth, and finally, our approach has minimal deployment requirements, which make it both practical and portable.

These advantages of our approach over existing techniques based on dynamic tainting are both conceptual (positive tainting and syntax-aware evaluation) and practical (accurate propagation and minimal deployment requirements). In the following sections we elaborate on the key features of our approach.

3.1 Positive Tainting

Positive tainting consists of the identification and marking of data coming from trusted sources. This approach contrasts with the conventional dynamic tainting approach, negative tainting, which is based on identifying and marking untrusted input sources. In this section, we discuss positive tainting and show how using positive instead of negative tainting has significant implications for the effectiveness of the overall approach.

One of the main problems with tainting-based approaches is the correct identification of all of the relevant data that should be marked. With negative tainting, incompleteness may generate false negatives, that is, it may leave an application vulnerable to attacks, and the problem may not be discovered until after an attack has actually occurred. With positive tainting, however, incompleteness in the identification of the data to be marked leads only to false positives that can be eliminated by the developer while keeping the system protected from attacks. Figure 2 shows a graphical depiction of this fundamental difference between the two tainting approaches.

In the context of SQL injection, this conceptual difference between positive and negative tainting is especially significant. The characteristics of SQLIAs make the complete identification of all sources of untrusted data problematic and, most importantly, the identification of trusted sources relatively straightforward. SQLIAs typically target Web applications, which are highly modular, deployed in many different configurations, and interface with a wide range of external systems. In these applications there are often many different sources of external input to be considered. Enumerating all of these potential sources of untrusted input is inherently error-prone and unsafe. Case in point, developers initially assumed that only user input needed to be marked as tainted. However, subsequent exploits demonstrated that additional sources of external input, such as browser cookies and uploaded files, also needed to be marked. Therefore, these input sources were added to the list of untrusted sources. Unfortunately, even this solution was not sufficient, as attackers soon realized they could use local server variables and the database itself as injection vectors. Even adding these additional sources is not guaranteed to provide a complete solution. If even a single source of malicious input is left unidentified, data from that source would be implicitly trusted, which would leave an application vulnerable to attacks.

The use of positive tainting addresses this problem. Identifying trusted, instead of untrusted, data in a Web application is in most cases straightforward because the set of trusted strings and trusted input sources that are used to build query strings are finite and typically small. Our approach works by first accurately and automatically identify all hard-coded strings in the application and marking these strings as trusted. (These strings have been defined by the developer and, thus, can be considered benign.) For the common case of applications in which developers create SQL queries by combining hard-coded strings that contain SQL keywords and operators with user inputs that are used only as literals, this set is complete (i.e., it includes all and only the trusted data). For these applications, our approach would accurately and fully automatically identify SQLIAs without generating any false positives. This is a common way of building Web applications and, incidentally, happened to be the case for all of the applications considered in our empirical study (see Section 5). In some applications, however, developers also use strings from external input to build keywords and operators in a query string. These strings should also be trusted because they are not hard-coded, they would not be part of the initial set of trusted data. In these situations, our approach would generate false-positives (but would never result in a vulnerability). To account for these cases, our approach provides developers with a mechanism to specify which additional data from which external sources should be trusted. The data sources can be of various nature, such as files, network connections, and server variables. This mechanism lets developers specify two aspects for each source: a trust marking and a trust policy. A trust marking is simply a unique id that our technique uses to distinguish data coming from the different sources. A trust policy characterizes the trustworthy data coming from a specific source and expresses legal ways in which it can be used. Trust policies are enforced during the syntax-aware evaluation, which we discuss in Section 3.3.

In a typical scenario, we expect developers to specify most of the trusted sources beforehand. However, some of them might be overlooked until after a false positive is reported. In this process, the set of trusted data sources grows monotonically and eventually converges to a complete set that produces no false positives. It is important to note that false positives that occur after deployment would be due to the use of information sources that have never been used in-house. In other words, false positives are likely to occur only for totally untested parts of the application. Therefore, even when developers failed to completely identify and mark additional sources of trusted input beforehand, we expect these sources to be identified during normal in-house testing of the application, and the set of trusted data to quickly converge to the complete set.

2We make the assumption that developers are trustworthy. In fact, an attack encoded by a developer would not be classified as an SQLIA, but rather as some form of back-door attack, which is not the problem addressed in this paper.
3.2 Accurate Taint Propagation

Taint propagation consists of maintaining the taint markings associated with the data in the program while these data are used and manipulated at runtime. When tainting is used for security-related applications, it is especially important to perform accurate propagation. Inaccurate propagation can undermine the effectiveness of a technique by allowing data to incorrectly lose or gain taint markings, which subsequently would cause it to be handled incorrectly. In our approach, we provide a mechanism to accurately mark and propagate taint information by (1) tracking the taint markings at a low level of granularity and (2) precisely accounting for the effect of functions that operate on the tainted data.

Character-level tainting. We track taint information at the character level rather than at the string level. We do this because, when building query strings, strings are constantly broken into substrings, manipulated, and combined. Tracking taint information at the string level would prevent our approach from precisely modeling the movement of tainted substrings. However, by associating taint information to single characters, we can accurately handle string-manipulation operations without introducing inaccuracies.

Accounting for string manipulations. To accurately maintain the character-level taint information, we must identify all relevant string operations and account for the effect of such operations on the marked characters (i.e., we must enforce complete mediation of all string operations). Our approach achieves this goal by taking advantage of the encapsulation offered by object-oriented languages, in which all string manipulations are performed using a small set of clearly defined classes and methods. Our approach extends all such classes and methods by adding to them functionality to keep track of taint markings based on the methods’ semantics.

We discuss the language specific details of our implementation of the taint markings and propagation in Section 4.

3.3 Syntax-Aware Evaluation

 Besides ensuring that taint markings are correctly created and maintained during execution, another issue is how to use the taint information to distinguish legitimate from malicious queries. One way to achieve this goal is to simply forbid the use of untrusted data in sensitive functions. This approach is not applicable in the context of SQL injection because it would identify as SQLIAs all queries that contains user input. Therefore, all queries built by combining predefined strings and user-provided input (i.e., the vast majority of the queries) would be prevented from executing.

To alleviate this problem, researchers have introduced the concept of declassification rules, which permit the use of tainted input as long as it has been processed by certain types of sanitizing functions. These functions are typically filters that perform various operations, such as regular expression matching or sub-string replacement. The idea of declassification is based on the assumption that these functions are able to eliminate or neutralize harmful parts of the input and to make the data safe. However, in practice, these checks are often insufficient and attacks can still occur even after input has passed through these sanitizing functions (i.e., these approaches often generate false negatives). Moreover, in the case of query building, inputs should be able to be used even if they have not been sanitized, as long as they are used in the right context within a query string.

Our approach addresses this issue by performing a syntax-aware evaluation of the queries before they are sent to the database. Syntax-aware evaluation considers the context in which trusted and untrusted data are used to make sure that all parts of a query other than string or numeric literals (e.g., SQL keywords and operators) consist only of trusted characters. As long as untrusted data is confined to literals, we are guaranteed that no SQLIA can be performed. Conversely, if the check reveals that there is, for instance, an SQL operator that contains characters not marked as trusted, we can assume that the operator has been injected by an attacker and block the query.

The use of syntax-aware evaluation instead of declassification rules frees us from having to make potentially unsafe assumptions about the types of sanitizing functions put in place by developers. It also allows for using untrusted input in a query, but ensures that when that input is used in the context of the query string, its usage is harmless for the database.

Our technique performs syntax-aware evaluation of a query string immediately before it is sent to the database to be executed. First, a customized database parser parses the query string and returns a set of tokens that are labeled as SQL keywords, operators, and literals and that correspond to sub-strings of the original query string. Then, the technique iterates through the set and checks whether all of the tokens identified as keywords or operators were constructed using only trusted data. It is fundamental that we use a database parser at this step because it ensures that our technique interprets the query in the same way as the database, thus avoiding problems with alternate encodings [1]—an attack method that obfuscates keywords and operators to avoid developer’s signature-based checks. If all of the keyword and operator tokens pass the check, the technique concludes that the query is safe and allows the application to send it to the database.

Note that this approach accounts also for cases in which external input is allowed to take the position of non-literals, that is, when an application uses external query fragments—parts of a query that are input from an external source. For example, Bugzilla (http://www.bugzilla.org) stores conditional clauses in a database that can then be loaded and reused at a later time. As discussed in Section 3.1, our approach allows developers to specify additional data from external sources that should be trusted. Developer can associate special trust markings and trust policies with these additional sources. In most cases, the default policy, which simply trusts all data from a given source, is enough. For example, such a policy can account for the common case in which parts of a query are stored (1) in external files that were created by the developers and reside on the server or (2) in database records that were created by the developers. During the syntax aware evaluation, strings originating from these sources are treated as trusted and can include SQL keywords and operators.

In other cases, additional trust policies may be needed. In particular, there may be cases in which only some specific sub-queries are expected from an external source. In these cases, the trust policy could specify legitimate forms (e.g., in the form of patterns) for query fragments that originate from a given external sources. When performing syntax-aware evaluation, our approach would then recognize the trust markings and evaluate specially-marked sub-queries according to the corresponding trust policy. In this way, only external fragments that comply with the developers’ specification would be accepted. This is especially useful in situations where developers want to allow data previously entered into the database to be used as parts of a query but want to prevent second-order injections [1]—injections of malicious strings into a database that result in an attack only when they are later used to build queries. In the case of Bugzilla (see above), developers may want to enforce the policy that all sub-query strings originating from the database are trusted only if they match a specific pattern.
They could, for example, use the pattern

```
{id:severity}='''w' (AND|OR) {id:severity}='''w'''
```

which specifies that all queries must build a conditional clause that involves only the `id` or `severity` fields and is connected using only AND or OR. In general, we expect the need for such additional policies to be fairly uncommon. In fact, we did not need to specify additional policies for any of the subjects we have considered or studied so far.

3.4 Minimal Deployment Requirements

Many existing approaches require the use of customized runtime systems or impose significant overhead on the protected applications (see Section 6). Conversely, our approach has the practical advantage of requiring minimal deployment infrastructure and imposing a low overhead on the protected applications. The use of our technique requires only minor, localized changes to the application’s bytecode to enable the usage of our modified string library and insert checks for the syntax-aware evaluation of queries. Furthermore, this process is fully automated and requires no developer intervention except in the cases where additional sources of trusted input must be added to the ones automatically identified. The modified application is deployed as a normal Web application along with our string library. Finally, the overhead introduced by the changes in the application is negligible. We discuss deployment requirements and overhead of the approach in greater detail in Sections 4.5 and 5.2.

4. IMPLEMENTATION

To be able to experiment with our approach, we have developed a prototype tool, called WASP (Web Application SQL Injection Preventer), that is written in Java and implements our technique for Java-based Web applications. We have chosen to target Java because it is a commonly used language for developing Web applications. Moreover, we already have a significant amount of analysis and experimental infrastructure for Java. We expect our approach to be generally applicable to other languages.

Figure 3 shows the high-level architecture of WASP. As the figure shows, WASP consists of a library (MetaStrings) and two core modules (string initializer and string checker). The MetaStrings library provides functionality for assigning trust markings to strings and precisely propagating the markings at runtime. Module string initializer enables the use of the MetaStrings library with the application using instrumentation. It identifies strings in the Web application, replaces them with corresponding MetaStrings classes, and suitably initialize trusted strings. Module string checker implements the syntax-aware evaluation discussed in Section 3.3. It adds to the code calls to a function that at runtime evaluates query strings before they are sent to the database.

In the subsequent sections we discuss the modules in more detail. We use the sample code introduced in Section 2 to provide illustrative examples of various implementation aspects.

4.1 The MetaStrings Library

MetaStrings is a library of classes that we developed that mimic and extend the behavior of Java’s string classes (i.e., `Character`, `String`, `StringBuilder`, and `StringBuffer`). For each string class C, MetaStrings provides a “meta” version of the class `MetaC` that has the same functionality as C, but allows for (1) associating sets of metadata to each character in a string, and (2) maintaining the metadata at runtime, as the string is manipulated.

In developing MetaStrings, we took advantage of the object-oriented characteristics of the Java language: encapsulation, information hiding, polymorphism, and dynamic binding. Encapsulation and information hiding guarantee that the internal representation of a string class is accessed only through the methods of the class. Polymorphism and dynamic binding let us add functionality to a string class by (1) creating a subclass that redfines all methods of the original class and (2) replacing instantiations of the original class with the subclass.

To illustrate, Figure 4 shows an intuitive view of a MetaStrings class that corresponds to Java’s `String` class. As the figure shows, `MetaString` extends class `String`, has the same internal representation, and provides the same methods. `MetaString` also contains additional data structures for storing metadata and for associating the metadata with characters in the string. Each method in `MetaString` overrides the corresponding method in `String`, provides the same functionality as the original method, and updates the metadata based on the method’s semantics. For example, a call to method `substring(2, 4)` on an instance `str` of the subclass would return a new instance of `MetaString` that contains only the second and third characters of `str` and that maintains the metadata associated with these two character in `str`. In addition to the overridden methods, MetaStrings classes also provide additional methods for setting and querying the metadata associated with a string’s characters.

The MetaStrings library’s propagation update policies specify a default behavior that propagates trust markings in accordance with the semantics of the corresponding string operations. However, MetaStrings also provides developers with a facility for specifying alternate propagation-update policies. This is particularly useful for specifying different ways that metadata should be maintained when certain string functions are called. For example, alternate propagation policies would be useful in the `String` method...
replace, which replaces all occurrences of a given character with a different specified character. In some cases, it might be desirable to have a replace function where the newly inserted character has the same trust markings as the replaced character. In other cases, developers may prefer not to implicitly transfer the trust markings, and leave the inserted character unmarked. In cases like these ones developers are able to specify which propagation semantics they want to use by defining a customized propagation-update policies within MetaStrings. (In our current implementation of MetaStrings, we have implemented the latter semantics for method replace.) In our evaluation of WASP there was no need to have alternate propagation update policies, but the functionality is nevertheless implemented in the current MetaStrings library, for future use and applications.

Overall, the use of MetaStrings has the following benefits: (1) allows for associating trust markings at the granularity level of the single characters; (2) accurately maintains and propagates trust markings; (3) is defined completely at the application level, and thus, does not require a customized runtime system; (4) only requires minimal changes to an application’s bytecode to be used (and those changes can be performed automatically, as discussed in Section 4.2); (5) imposes low overhead on the application (See Section 5.2).

The main limitation of the MetaStrings library’s implementation is that it currently cannot assign, maintain or propagate trust markings if an application either uses array of characters to represent and manipulate strings or manipulates strings within native methods. In practice, these limitations are of little relevance. Representing strings using arrays instead of string classes is inconvenient and very rarely done. Moreover, during the instrumentation of the Web application, we identify and report these potentially problematic situations to the developers. It is worth noting that none of these situations occurred for the subject applications considered in our empirical evaluation (see Section 5).

4.2 Initialization of Trusted Strings

To implement our positive tainting approach described in Section 3.1, WASP must be able to identify trusted strings and mark them. There are three categories of strings that WASP must consider: hard-coded strings, strings implicitly created by Java, and strings originating from external sources. The way in which a string is instrumented and marked depends on the category to which it belongs, as we explain in the following sections.

**Hard-Coded Strings.** The identification of hard-coded strings in an application’s bytecode is a fairly straightforward process. In Java, hard-coded strings are represented using `String` objects that are created automatically by the Java Virtual Machine (JVM) when string literals are loaded onto the stack. (The JVM is a stack-based interpreter.) Therefore, to identify hard-coded strings, WASP simply scans the bytecode and identifies all load instructions whose operand is a string constant. WASP then instruments the code by adding, after each of these load instructions, code that creates an instance of the `MetaString` class using the hard-coded string as a parameter. Finally, because hard-coded strings are completely trusted, WASP adds to the code a call to the method of the newly created `MetaString` object that marks all characters as trusted.

At runtime, polymorphism and dynamic binding allow this instance of the `MetaString` object to be used in any place where the original `String` object would have been used.

Figure 4.2 shows an example of this bytecode transformation. The Java code at the top of the figure is Line 4 of our servlet example (see Figure 1). It creates one of the hard-coded strings in the servlet. Underneath, we show the original bytecode on the left, and
the modified bytecode on the right. The modified bytecode con-
tains the additional instructions that (1) load a new MetaString on the stack, (2) call the MetaString constructor using the previous string as an initialization parameter, and (3) call the method markAll, which assigns the given trust marking to all characters in the string.

Implicitly-Created Strings. In Java programs, the creation of some string objects is implicitly added by the compiler. For example, Java compilers typically translate the string concatenation operator ("+") into a sequence of calls to the append method of a newly created StringBuilder class. WASP must convert these string objects into their corresponding MetaStrings objects, so that they can maintain and propagate the trust markings of the strings on which they operate. To do this, WASP scans the bytecode for instructions that create new instances of the string classes used to perform string manipulation (e.g., StringBuilder). WASP then modifies each such instruction so that it creates an instance of the corresponding MetaStrings class. In this case, WASP does not associate any trust markings with the newly-created MetaStrings objects; these objects are not trusted per se and the strings (or parts thereof) they contain become marked only if the actual values assigned to them during execution are marked.

Figure 6 illustrates the instrumentation added by WASP for internal strings. The Java source code corresponds to line 5 in our example servlet. The StringBuilder object at offset 28 in the original bytecode is created by the Java compiler when translating the string concatenation operator ("+"). WASP replaces the instantiation at offset 28 with the instantiation of a MetaStringBuilder class and then changes the subsequent invocation of the constructor at offset 37 so it matches the newly instantiated class. Because MetaStringBuilder extends StringBuilder, the subsequent calls to the append method invoke the correct method in the MetaStringBuilder class.

Strings from External Sources. To allow additional sources of trusted input to be used, we allow developers to list them, and related customized trust policies, in a configuration file. Developers can specify various types of sources, such as files (by name), network connections (by host), and database servers (by database name, table, field, or a combination thereof). For each source, developers can specify a specific trust marking that corresponds with a specific trust policy or simply assign the default trust marking that WASP assigns to hard-coded strings. For space reasons, we only discuss the implementation for one specific source—trusted external files.

Consider the situation of an application in which the developer stores various query fragments in an external file. In the configuration file, the developer specifies the name of the file (e.g., foo.txt) as a trusted source of strings. Based on this information, WASP scans the bytecode for all instantiations of new file objects (i.e., FileInputStream, FileReader) and instruments them with code that checks the name of the file being opened. At runtime, if the name of the file matches the name(s) specified by the developer (foo.txt in our case), the file object is added to an internal list of currently-trusted file objects. WASP also instrument all calls to methods of file stream objects that return strings, such as BufferedReader.readLine. At runtime, the added code checks to see whether the object on which the method is called is in the list of currently trusted file objects. If so, it marks the generated strings with the trust marking specified by the developer for the corresponding source.
We use a similar strategy to mark network connections. In this case, instead of matching file names at runtime, we match hostnames. The interaction with databases is more complicated and requires that WASP not only matches the initiating connection, but also traces tables and fields through instantiations of the `Statement` and `ResultSet` objects created when querying the database.

Although functional, the described implementation may create unneeded instrumentation. We can limit the amount of instrumentation inserted in the code by leveraging static information about the program. For example, data-flow analysis could identify strings that are not involved at all with the construction of query strings and, thus, need not be instrumented. For another example, if WASP can statically determine that the filename of a given file object is not one of the trusted filenames specified by the developer, then that object would not need to be instrumented. Analogous optimizations could be implemented for other external sources. We did not incorporate any of these optimizations in the current tool because we were mostly interested in having an initial prototype to assess our technique. However, we are planning to implement them in future work.

### 4.3 Handling False Positives

When presenting our approach, in Section 3, we discussed how developers can specify external sources of trusted data either beforehand or to eliminate false positives. To assist developers in eliminating false positives, WASP can operate in "learning mode." The learning mode would be typically used during in-house testing and consists of associating with strings an additional set of markings in addition to the trust marking discussed above. (As shown in Figure 4, MetaStrings can associate a set of markings to each character in a string, so trust markings and the additional set of markings can coexist.) These additional markings are applied to all string objects and help tracing back a string to the point in which it was created.

Currently, the marking for the characters in a string $s$ is a unique ID that maps to the point of creation of the string (expressed as a fully qualified class name, a method signature, and a bytecode offset). If WASP detects an SQLIA while in learning mode, it uses the markings associated with the untrusted SQL keywords and operators in the query to report the position in the code where the corresponding strings were created. If the SQLIA is actually a false positive, knowing the position in the code of the offending string(s) can help developers identify the cause of the detection and correct omissions in the set of trusted inputs.

Although it is only partially implemented, we are currently adding to WASP a feature that detects when an offending set of characters is first used in an application. By leveraging this feature, developers will be able to specify via WASP's configuration file the literal character with full trust markings associated with it. Because the default trust policy is that all keyword and operator tokens must have originated from trusted strings, WASP considers them to be safe and allows the query to be executed on the database. Otherwise, WASP classifies the query as an SQLIA, prevents it from executing, and reports it.

We illustrate how the syntax-aware evaluation works using our example servlet and the legitimate and malicious query examples from Section 2. For the servlet, there are no external sources of strings, so WASP only marks as trusted the hard-coded strings, and only the default trust policy is applied. Figure 7 shows the sequence of tokens in the legitimate query as they would be parsed by the MetaChecker. In the figure, SQL keywords and operators are surrounded by boxes. The figure also shows the trust markings associated with the strings, where an underlined character is a character with full trust markings associated with it. Because the default trust policy is that all keyword and operators tokens must have originated from trusted strings, MetaChecker simply checks whether all these tokens are comprised of trusted characters. This query conforms to the trust policy and is thus allowed to execute on the database.

Consider now the malicious query, where the attacker submits a login of "admin", "a" and a random pin of "0." Figure 8 shows the sequence of tokens for the resulting query together with the trust markings. Recall that the -- operator is the SQL comment character, so everything after this operator is identified by the parser as pure text. In this case, the MetaChecker would find that the last two tokens, and contain untrusted characters. It would therefore classify the query as an SQLIA, prevent it from executing, and report it.

### 4.4 Syntax-Aware Evaluation

The string checker module performs two tasks: it instruments the application bytecode to insert calls that perform the syntax-aware evaluation, and at runtime, parses and checks the query strings.

To instrument the application, WASP first identifies all of the `database interaction points`: points in the application where query strings are issued to an underlying database. In Java, all calls to the database are performed via specific methods and classes in the JDBC library (`http://java.sun.com/products/jdbc`). Therefore, these points can be identified through a simple matching of method signatures. After identifying the database interaction points, WASP inserts a call to the syntax-aware evaluation function, called MetaChecker, right before the interaction point. The evaluation function takes as a parameter the MetaStrings object that contains the query about to be executed.

At runtime, WASP performs the syntax-aware evaluation as described in Section 3.3. The MetaChecker first tokenizes the query string using a parser that understands the developer-specified dialect of SQL used by the application. To implement the default trust policy, it iterates through the keyword and operator tokens and examines their trust markings. If any of these tokens are not marked as trusted, the query is blocked and reported.

If there are custom marking on the token, WASP checks them against the corresponding developer-specified trust policy. In general, custom policies could range from simple pattern matching to more complex analyses that take advantage of additional context information. In our current implementation, WASP accepts policies specified as regular expressions over the set of marked tokens. If all tokens are compliant with their custom trust policies, WASP considers them to be safe and allows the query to be executed on the database. Otherwise, WASP classifies the query as an SQLIA, prevents it from executing, and reports it.

### 4.5 Deployment Requirements

Using WASP to protect a Web application requires the developer to run an instrumented version of the application. There are two general implementation strategies that we can follow when performing the instrumentation: off-line or on-line. Off-line instrumentation instruments the application statically and deploys the instrumented version of the application. On-line instrumentation, conversely, deploys an unmodified application (except for some bootstrap code) and modifies the code on the fly, at load time (i.e. when classes are loaded by the JVM). This latter option allows for a great deal of flexibility and can be implemented by leveraging the new instrumentation package introduced in Java 5 (`http:...`)
Unfortunately, the current implementation of the Java 5 instrumentation package offers only partial functionality and does not yet provide some key features needed by WASP. In particular, it does not allow for clearing the final flag in the string library classes, which prevents the MetaStrings library from extending them. Because of this limitation, in the current implementation of WASP, we have chosen to rely on off-line instrumentation and to splice into the Java library a version of the string classes in which the final flag has been cleared.

Overall, the deployment requirements for our approach are very lightweight. The modification of the Java library is performed only once, in a fully automated way, and takes just a few seconds. No modification of the Java Virtual Machine is required. The instrumentation of the Web application is also performed automatically: WASP instruments the application and creates a deployment archive that contains the instrumented application, the MetaStrings library, and the string checker module. At this point, the archive can be deployed just like any other Web application.

5. EVALUATION

The goal of our empirical evaluation is to assess the effectiveness and efficiency of the approach presented in this paper when applied to a testbed of Web applications. We used our implementation of WASP to perform the evaluation. We investigated the following three research questions:

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

**RQ2:** What percentage of legitimate accesses does WASP identify as SQLIAs and prevent from executing on the database?

**RQ3:** How much runtime overhead does WASP impose?

The first two questions deal with the **effectiveness** of the technique: RQ1 addresses the false negative rate of the technique, and RQ2 addresses the false positive rate. RQ3 deals with the **efficiency** of the proposed technique. The following sections explain our setup and protocol for the evaluation and discuss the results of the studies we performed to answer the research questions.

5.1 Experiment Setup

Our experiments are based on the evaluation framework used for AMNESIA [9] and SQLCheck [22]. This framework provides a testbed that consists of Web applications, logging infrastructure, and a large set of test cases containing both legitimate inputs and SQLIAs. In the next two sections we summarize the relevant details of this framework.

5.1.1 Subjects

The set of subjects consists of seven Web applications that accept user input via Web forms and use it to build queries to an underlying database. Five of the seven applications are commercial applications that we obtained from GotoCode (http://www.gotocode.com/): Employee Directory, Bookstore, Events, Classifieds, and Portal. The other two, Checkers and OfficeTalk, are student-developed applications that have been used in previous related studies [7].

Table 1 provides, for each subject, the size in terms of lines of code (LOC) and the number of database interaction points, (DBIs). Overall, the deployment requirements for our approach are very lightweight. The inputs were generated independently by a Master’s level student with experience in developing commercial penetration testing tools for Web applications. Test cases were not generated for non-accessible servlets and for state parameters.

Table 1. Evaluation subjects. The number of injectable parameters (Params) in parenthesis. The non-injectable parameters are state parameters, whose purpose is to maintain state, and which are not used to build queries.

<table>
<thead>
<tr>
<th>Subject</th>
<th>LOC</th>
<th>Params</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee Directory</td>
<td>870</td>
<td>3</td>
</tr>
<tr>
<td>Bookstore</td>
<td>492</td>
<td>3</td>
</tr>
<tr>
<td>Events</td>
<td>446</td>
<td>3</td>
</tr>
<tr>
<td>Classifieds</td>
<td>738</td>
<td>3</td>
</tr>
<tr>
<td>Checkers</td>
<td>320</td>
<td>3</td>
</tr>
<tr>
<td>OfficeTalk</td>
<td>318</td>
<td>3</td>
</tr>
</tbody>
</table>

5.1.2 Test Case Generation

For each application in the testbed there are two sets of inputs: **LEGIT**, which consists of legitimate uses of the application, and **ATTACK**, which consists of SQLIAs. The inputs were generated independently by a student with experience in developing commercial penetration testing tools for Web applications. Test cases were not generated for non-accessible servlets and for state parameters.

To create the ATTACK set, the student first built a set of potential attack strings by surveying different sources: (1) exploits developed by professional penetration-testing teams to take advantage of SQL-injection vulnerabilities; (2) online vulnerability reports that included US-CERT (http://www.us-cert.gov/) and CERT/CC Advisories (http://www.cert.org/advisories/); and (3) information extracted from several security-related mailing lists. The resulting set of attack strings contained thirty unique attacks that had been used against applications similar to those in the testbed. All types of attacks reported in literature [10] were represented in this set except for multi-phase attacks such as overly-descriptive error messages and second-order injections. Since multi-phase attacks require human intervention and interpretation, we omitted them to keep our testbed fully automated. The student then generated a complete set of inputs for each servlet’s injectable parameters using values from the initial attack strings and legitimate values. The result was a broad set of potential SQLIAs. Because many of the applications performed sufficient input validation and could block a portion of the attacks, the student eliminated the blocked attacks from the set. We used the remaining attacks as the ATTACK set.

The LEGIT set was created in a similar fashion. However, instead of using attack strings to generate sets of parameters, the student used legitimate values. To create “interesting” legitimate values, we asked the student to create inputs that would stress and possibly break naive SQLI detection techniques (e.g., techniques based on simple identification of keywords or special characters in the input). The result was a set of legitimate inputs that contained...
Table 1: Subject programs for the empirical study.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Description</th>
<th>LOC</th>
<th>DBAs</th>
<th>Servlets</th>
<th>Params</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkers</td>
<td>Online checkers game</td>
<td>5,421</td>
<td>5</td>
<td>18 (61)</td>
<td>44 (44)</td>
</tr>
<tr>
<td>Office Talk</td>
<td>Purchase-order management system</td>
<td>4,543</td>
<td>40</td>
<td>7 (64)</td>
<td>13 (14)</td>
</tr>
<tr>
<td>Employee Directory</td>
<td>Online employee directory</td>
<td>5,658</td>
<td>25</td>
<td>7 (10)</td>
<td>25 (34)</td>
</tr>
<tr>
<td>Bookstore</td>
<td>Online bookstore</td>
<td>16,959</td>
<td>71</td>
<td>8 (28)</td>
<td>36 (42)</td>
</tr>
<tr>
<td>Events</td>
<td>Event tracking system</td>
<td>7,202</td>
<td>31</td>
<td>13 (12)</td>
<td>36 (46)</td>
</tr>
<tr>
<td>Classifieds</td>
<td>Online management system for classifieds</td>
<td>10,589</td>
<td>34</td>
<td>6 (14)</td>
<td>18 (26)</td>
</tr>
<tr>
<td>Portal</td>
<td>Portal for a club</td>
<td>16,453</td>
<td>67</td>
<td>3 (28)</td>
<td>39 (46)</td>
</tr>
</tbody>
</table>

Table 2: Results for RQ1 and RQ2.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Unsuccessful</th>
<th>Attack Successful</th>
<th>Detected</th>
<th>Blkd.</th>
<th>Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkers</td>
<td>5,394</td>
<td>499</td>
<td>499</td>
<td>0</td>
<td>424</td>
</tr>
<tr>
<td>Office Talk</td>
<td>4,332</td>
<td>2,066</td>
<td>2,066</td>
<td>0</td>
<td>660</td>
</tr>
<tr>
<td>Empl. Dir.</td>
<td>4,155</td>
<td>1,999</td>
<td>1,999</td>
<td>0</td>
<td>608</td>
</tr>
<tr>
<td>Bookstore</td>
<td>4,066</td>
<td>2,141</td>
<td>2,141</td>
<td>0</td>
<td>900</td>
</tr>
<tr>
<td>Events</td>
<td>3,995</td>
<td>1,973</td>
<td>1,973</td>
<td>0</td>
<td>568</td>
</tr>
<tr>
<td>Classifieds</td>
<td>3,387</td>
<td>3,016</td>
<td>3,016</td>
<td>0</td>
<td>1,080</td>
</tr>
</tbody>
</table>

Table 3: Absolute overhead (in seconds) imposed by WASP.

<table>
<thead>
<tr>
<th>Subject</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkers</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.0009</td>
</tr>
<tr>
<td>Office Talk</td>
<td>0.004</td>
<td>0.012</td>
<td>0.0012</td>
<td>0.0012</td>
<td>0.0012</td>
<td>0.0010</td>
</tr>
<tr>
<td>Empl. Dir.</td>
<td>0.0379</td>
<td>0.0461</td>
<td>0.0445</td>
<td>0.0468</td>
<td>0.0451</td>
<td>0.0441</td>
</tr>
<tr>
<td>Bookstore</td>
<td>0.0114</td>
<td>0.0219</td>
<td>0.0217</td>
<td>0.0207</td>
<td>0.0215</td>
<td>0.0195</td>
</tr>
<tr>
<td>Events</td>
<td>0.0044</td>
<td>0.0092</td>
<td>0.0090</td>
<td>0.0090</td>
<td>0.0085</td>
<td>0.0080</td>
</tr>
<tr>
<td>Classifieds</td>
<td>0.0050</td>
<td>0.0104</td>
<td>0.0108</td>
<td>0.0104</td>
<td>0.0103</td>
<td>0.0090</td>
</tr>
<tr>
<td>Portal</td>
<td>0.0829</td>
<td>0.0809</td>
<td>0.0791</td>
<td>0.0848</td>
<td>0.0795</td>
<td>0.0832</td>
</tr>
</tbody>
</table>

SQL keywords, operators, and troublesome characters such as single quotes and comment operators.

5.2 Results

To address RQ1 and RQ2, we ran the ATTACK and LEGIT input sets against the tested applications and measured the percentage of attacks reported by WASP. Table 2 shows the results. For RQ1, we ran all of the inputs in the ATTACK set against the instrumented applications. We recorded the total number of successful attacks in the set (Successful), the number of unsuccessful (i.e., stopped by the application itself) attacks in the set (Unsuccessful), and the number of attacks that WASP detected (Detected). For RQ2 we ran all of the inputs in the LEGIT set and recorded the number of attacks blocked by WASP (Blkd.) and the number of legitimate inputs WASP allowed to execute (Allowed).

As the table shows, for RQ1, WASP did not generate any false negatives—it was able to detect and prevent all of the attacks that the application did not block on its own. Similarly, for RQ2, WASP achieved a perfect score and did not generate any false positives—it allowed all of the legitimate queries corresponding to the inputs in the LEGIT set to execute on the database.

To address RQ3, we measured the overhead imposed by WASP on the subjects. We first ran the inputs in the LEGIT set against the uninstrumented applications. Then, we reran all the inputs against the applications instrumented by WASP. In both cases we measured the total time required for the application to return a Web page after an input had been submitted. The difference between the two measurements represents the overhead imposed by WASP. We performed the study on a dedicated Pentium 4, with 1GB of memory, running the GNU/Linux Operating System, and using a database running on the same host. We repeated all measurements five times and averaged the results to limit the risk of imprecision introduced by external effects. Table 3 shows the results of the study. For each of the seven subjects, the table reports the absolute time overhead in seconds per servlet access for the five runs and the overall average.

In most cases the runtime overhead imposed by WASP is negligible. With the exception of one application, the overhead ranges from .001s to .0441s. For most Web applications, this cost would be dominated by the cost of the network access and database transaction. One application, Portal, incurred an overhead considerably higher than the other applications. After further investigation, we found that the overhead is due to an inefficiency in WASP’s implementation. Portal generates a huge number of string-based lookup tables and, although these strings are not used to build queries, WASP associates (and propagates) trust marks for all these strings. Some of the optimizations discussed in Section 4.2 would eliminate this issue and considerably reduce the overhead measured for Portal. More generally, the study was performed using a completely unoptimized prototype of WASP, so the current results can be considered an upper bound for the cost of our approach.

5.3 Discussion of Results

The results of our studies indicate that WASPs is a viable technique for preventing and detecting SQLIAs. In our evaluation, WASP was able to correctly identify all SQLIAs without generating any false positives. WASP stopped 12,616 otherwise successful SQLIAs and correctly allowed 5,309 legitimate usages. Moreover, these results were achieved with an absolute overhead of less than .045s in most cases—a negligible overhead that would be dominated by network and database access times.

In our study, we have identified two potential threats to the generalizability of our results. First, the applications in the testbed might not be representative of Web applications in general. However, all but two applications are commercial applications and were used in two other studies. Second, the attacks generated might not be representative of actual attacks. To minimize this risk, we employed the services of a Master’s level student with experience in SQLIAs, penetration testing tools, and Web scanners, but not familiar with our technique. Also, the generated attack strings were based on real world attacks.

6. RELATED WORK

The use of dynamic tainting to prevent SQLIAs has been proposed by several researchers. The two approaches most similar to ours are those by Nguyen-Tuong and colleagues [19] and Pietraszek and Berghel [20]. Similar to them, we track taint information at the character level and use a syntax-aware evaluation to examine
tainted input. However, our approach differs in several important aspects. We introduce the use of positive tainting, which, as we discussed in Section 3.1, is an inherently safe way of identifying trusted data. Additionally, we improve on the idea of syntax-aware evaluation by (1) using a database parser to interpret the query string before it is executed, so ensuring that our approach can handle attacks based on alternate encodings, and (2) providing a flexible mechanism that allows different trust policies to be associated with different input sources. A practical advantage of our approach is that it has lightweight deployment requirements. Their approaches, conversely, require the use of an extensively customized PHP runtime interpreter, which adversely affects practical applicability and portability of the approaches.

Other dynamic tainting approaches more loosely related to our approach are those by Haldar, Chandra, and Franz [8] and Martin, Livshits, and Lam [16]. Although they also propose dynamic tainting approaches for Java-based applications, their techniques differ significantly from ours. First, they track taint information at the string level of granularity, which introduces imprecision in modeling string operations. Second, they use classification rules, instead of syntax-aware evaluation, to assess whether a query string contains an attack. Classification rules assume that sanitizing functions are always effective and are inherently unsafe—in many cases, attack strings can pass through sanitizing functions and still be harmful. Another dynamic tainting approach, proposed by Newsome and Song [18], focuses on tainting at a level that is too low to be used for detecting SQLIAs and is impractical in terms of execution overhead.

Researchers have also proposed dynamic techniques against SQLIAs that do not rely on tainting. These techniques include Intrusion Detection Systems (IDS) and automated penetration testing tools. Scott and Sharp propose Security Gateway [21], which uses developer-provided rules to filter Web traffic, identify attacks, and apply preventive transformations to potentially malicious inputs. The success of this approach depends on the ability of developers to write accurate and meaningful filtering rules. Similarly, Valeur and colleagues [23] developed an IDS that uses machine learning to distinguish legitimate and malicious queries. However, this technique is limited by the quality of the training set used to train the IDS. Machine learning was also used in WAVES [12], an automated penetration testing tool that probed websites for vulnerability to SQLIAs. Like all testing tools, WAVES cannot provide any guarantees of completeness. SQLrand [2] appends a random token to SQL keywords and operators in the application code. A proxy server then checks to make sure that all keywords and operators contain this token before sending the query to the database. Because the SQL keywords and operators injected by an attacker would not contain this token, they would be easily recognized as attacks. The drawbacks of this approach are that the secret token could be guessed, so rendering the approach ineffective, and that the approach requires the deployment of a special proxy server.

Model-based approaches against SQLIAs include AMNESIA [9], SQL-Check [22], and SQLGuard [3]. AMNESIA, previously developed by two of the authors, combines static analysis and runtime monitoring to detect SQLIAs. The approach uses static analysis to build models of the different types of queries an application can generate and dynamic analysis to intercept and check the query strings generated at runtime against the model. Non-conforming queries are identified as SQLIAs. Problems with this approach are that it is dependent on the precision and efficiency of its underlying static analysis and is not likely to scale to large applications. Our new technique takes a purely dynamic approach to preventing SQLIAs, so eliminating scalability and precision problems. In [22], Su and Wassermann present a formal definition of SQLIAs and propose a sound and complete (under certain assumptions) algorithm that can identify all SQLIAs by using an augmented grammar and by distinguishing untrusted inputs from the rest of the strings by means of a marking mechanism. This is an effective approach whose main weakness resides in its practical applicability. The approach requires the manual intervention of the developer to identify and annotate untrusted sources of input, which introduces completeness problems and may lead to false negatives. Our use of positive tainting eliminates this problem while providing similar guarantees in terms of effectiveness. SQLGuard [3] is an approach similar to SQLCheck. The main difference is that SQLGuard builds its models on the fly by requiring developers to call a special function and to pass to the function the query string before user input is added.

Other approaches against SQLIAs rely purely on static analysis. These approaches scan the application and leverage information flow analysis or heuristics to detect code that could be vulnerable to SQLIA. Livshits and Lam [14] and WebSSARI [13] use information flow analysis to determine if user input can reach sensitive functions in an application. If so, a potential vulnerability is reported to the developer. Because of the imprecise nature of the static analysis they use, these techniques can generate false positives. Moreover, since they rely on classification rules to transform untrusted input into safe input, they can generate false negatives. Wassermann and Su propose a technique [24] for detecting if an application can generate queries that contain tautologies by combining static analysis and automated reasoning. This technique is, by definition, limited in the types of SQLIAs that it can detect.

Finally, researchers have also focused on ways to directly improve the code of an application and eliminate vulnerabilities. Defensive coding best practices [11] have been proposed to eliminate SQL injection vulnerabilities. However, coding practices are in general ineffective because they mostly rely on the ability and training of the developer. Moreover, there is much literature on how to evade certain types of defensive-coding practices, including “pseudo-remedies” such as stored procedures and prepared statements (e.g., [1, 15, 11]). Researchers have also developed special libraries that can be used to safely create SQL queries [4, 17]. These approaches, however, require developers to learn new APIs for developing queries, are not easily applicable to existing code, and limit the expressiveness of SQL. Finally, JDBC-Checker [6, 7] is a static analysis-based tool that detects potential type mismatches in dynamically generated queries. Although it was not intended to prevent SQLIAs, JDBC-Checker can be effective against those SQLIAs that leverage vulnerabilities due to type-mismatches.

7. CONCLUSION

In this paper we have presented a novel, highly automated approach for detecting and preventing SQL injection vulnerabilities in Web applications. Our approach is based on (1) identifying trusted data sources and (2) allowing only data from such sources to become a SQL keyword or operator in query strings. Unlike previous approaches, our technique of positive tainting explicitly identifies trusted (rather than untrusted) data in the program. In this way, we eliminate the problem of false negatives that may result from the incomplete identification of all untrusted data sources. False positives, while possible in some cases, can typically be easily eliminated through testing. Our approach also provides practical advantages over the many existing techniques whose application requires customized and complex runtime environments. Our approach is defined at the application level, it requires no modification of the runtime system (JVM), and it imposes a low execution overhead.
We have evaluated our approach by developing a prototype tool, WASP, and using the tool to protect several applications when subjected to a large and varied set of attacks and legitimate accesses. WASP successfully and efficiently stopped over 12,000 attacks without generating any false positives.

We have three immediate goals for future work. The first goal is to further improve the efficiency of the technique, by leveraging analysis information to reduce the amount of instrumentation required. The second goal is to implement the approach for binary programs, by leveraging binary instrumentation infrastructures. Finally, we plan to evaluate our technique in a completely realistic context, by protecting one of the Web applications running at Georgia Tech with WASP and assessing the effectiveness of WASP in stopping real attacks directed at the application while allowing legitimate accesses.

Acknowledgments
This work was partially supported by DHS contract FA8750-05-2-0214 and NSF award CCR-0209322 to Georgia Tech.

8. REFERENCES