Sania: Syntactic and Semantic Analysis for Automated Testing against SQL Injection

Yuji Kosuga†, Kenji Kono†, Miyuki Hanaoka†
Department of Information and Computer Science
†Keio University
3-14-1 Hiyoshi Kohoku-ku, Yokohama, Japan
{yuji, hanayuki}@sslab.ics.keio.ac.jp, kono@ics.keio.ac.jp

Miho Hishiyama‡, Yu Takahama‡
‡IX Knowledge Inc.
3-22-23 Kaigan Minato-ku, Tokyo, Japan
{miho.hishiyama, takahama}@ikic.co.jp

Abstract

With the recent rapid increase in interactive web applications that employ back-end database services, an SQL injection attack has become one of the most serious security threats. The SQL injection attack allows an attacker to access the underlying database, execute arbitrary commands at intent, and receive a dynamically generated output, such as HTML web pages. In this paper, we present our technique, ***Sania****, for detecting SQL injection vulnerabilities in web applications during the development and debugging phases.* ***Sania*** *intercepts the SQL queries between a web application and a database, and automatically generates elaborate attacks according to the syntax and semantics of the potentially vulnerable spots in the SQL queries. In addition,* ***Sania*** *compares the parse trees of the intended SQL query and those resulting after an attack to assess the safety of these spots. We evaluated our technique using real-world web applications and found that our solution is efficient in comparison with a popular web application vulnerabilities scanner. We also found vulnerability in a product that was just about to be released.*

1 Introduction

The recent surge in the growth of the Internet has resulted in the offering of a wide range of web services, such as on-line stores, e-commerce, social network services, etc. However, web applications designed to interact with back-end databases are threatened by SQL injection attacks. SQL injection is a technique maliciously used to obtain unrestricted access to databases by inserting maliciously crafted strings to SQL queries via a web application. It allows an attacker to spoof his identity, expose and tamper with existing data in databases, and control databases server with the privileges of its administrator. It is caused by a semantic gap in the manipulation of user inputs between a database and a web application. Although a web application handles the user inputs as a simple sequence of characters, a database handles them as query strings and interprets them as a meaningfully structured command. According to Armorize Technologies [3], 376 SQL injection vulnerabilities were reported in 2006, making up 28% of all the reported vulnerabilities for web applications. The common vulnerability and exposures (CVE) hosted by Mitre [20] also reported that it was making up 14% of all those reported.

Sanitizing is a technique that can be used to prevent SQL injection attacks by escaping potentially harmful characters in client request messages. Suppose that a database contains `name` and `password` fields in the `users` table, and a web application contains the following code to authenticate a user's log in.

```
sql = "SELECT * FROM users WHERE name = '"
    + request.getParameter(name)
    + "' AND password = '"
    + request.getParameter(password) + "'";
```

This code generates a query to obtain the authentication data from a relational database. If an attacker inputs "`' or '1'='1`" into the `password` field, the query becomes:

```
SELECT * FROM users WHERE
 name = 'xxx' AND password = '' or '1'='1'
```

The `WHERE` clause of this query is always evaluated to be true, and thus an attacker can bypass the authentication, re-

gardless of the data inputted in the `name` field. To prevent this SQL injection, the web application must sanitize every single quote by replacing it with double quotes. If sanitizing is done appropriately, the query becomes:

```
SELECT * FROM users WHERE
 name = 'xxx' AND password = '" or "1"="1'
```

where the inputted values are regarded as a string. This technique prevents an SQL injection from changing the syntax of SQL.

Although sanitizing all the vulnerable spots is a sufficient measure for preventing SQL injection attacks, it is the plain truth that vulnerabilities may still remain even if a skilled and educated programmer writes the sanitation codes. This is because sanitizing is typically done by hand without any supporting tools.

Existing automated tools for discovering SQL injection vulnerabilities are based on a penetration testing that indiscriminately applies malicious codes to every injectable parameter. They check the contents returned from a web application to determine whether an SQL injection attack is successful or not. However, they mistakenly judge an attack as successful if the application returns a response different from an innocent one, although the SQL injection itself failed. For example, the response pages contents are always different when a web application inserts some data from a client into its database, and then the web application generates and sends a response page using the data that has just been inserted into the database. Moreover, these tools try to inject malicious code in brute force and thus take longer time to check for vulnerabilities.

Sania is designed to be used in the development and debugging phases. Thus, **Sania** can investigate HTTP requests and SQL queries to try to discover SQL injection vulnerabilities. By investigating SQL queries, **Sania** automatically identifies potentially vulnerable spots in the SQL queries, generates attack codes to attempt to exploit the vulnerable spots, and checks whether the web application is vulnerable.

To identify the vulnerable spots, **Sania** analyzes the SQL queries issued in response to the HTTP requests, and discovers the vulnerable spots in SQL queries in which an attacker can insert arbitrary strings. Then, **Sania** generates attack requests based on the *context* of the potentially vulnerable spots in the SQL queries. Since **Sania** generates attack requests using the context in which the vulnerable spots appear, it can also generate context-dependent attacks. For example, **Sania** generates an attack request that exploits two vulnerable spots at the same time if they are adjacent in the query and their types are both string expressions. To verify a web application, **Sania** compares the parse trees of the SQL queries. If the tree generated from an innocent HTTP request differs from that generated from an attack request, **Sania** determines that there is an SQL injection vulnerability. When compared with the existing tools, **Sania** is expected to generate fewer but more precise attacks because it takes the context of the vulnerable spots into consideration, but will detect more vulnerabilities. Existing tools, which generate attacks using brute force, generate more attacks but cannot cover all the SQL injection possibilities. In addition, **Sania** can more precisely detect vulnerabilities because it checks if an attack has succeeded or not by verifying the parse tree of the issued SQL.

We evaluated **Sania** using six real-world web applications. **Sania** was proved to be efficient, finding 39 vulnerabilities and causing only 13 false positives. Paros [23], a popular scanner for web application vulnerabilities, found only 5 vulnerabilities and caused 67 false positives under the same evaluation conditions. Moreover, we tested a production-quality web application using **Sania**. The tested application was in the final testing phase before being shipped to the customer. **Sania** successfully found one vulnerability in the application.

The remainder of this paper is organized as follows. We begin the next section by reviewing and discussing related work. Section 3 presents **Sania** and Section 4 describes the implementation. Section 5 presents our experimental results. Finally, we conclude the paper in Section 6.

2 Related Work

In this section, we list the work closely related to ours and discuss their pros and cons. The techniques related to SQL injection are classified and were evaluated by Halfond *et al.* [10], and we also made our classifications based on it. Interested readers can refer to [28] for the formal definition of SQL injection, and to [1, 2, 19, 25, 26] for the attacking and preventing techniques.

Framework Support Recent frameworks for web applications provide a functionality that can be used to prevent SQL injections. For example, Struts [27] supports a *validator*. A validator verifies an input from the user conforms to the pre-defined format of each parameter. If a validator prohibits an input from including meta-characters, we can avoid SQL injections. Since a validator does not transform the dangerous characters to safe ones, we can not prevent SQL injections if we want to include meta-characters in the input.

Prepare Statement SQL provides the `prepare` statement, which separates the values in a query from the structure of SQL. The programmer defines a skeleton of an SQL query and then fills in the holes of the skeleton at runtime. The `prepare` statement makes it harder to inject SQL

queries because the SQL structure can not be changed. Hibernate [11] enforces us to use the `prepare` statement. To use the `prepare` statement, we must modify the web application entirely; all the legacy web applications must be re-written to reduce the possibility of SQL injections. **Sania** is useful to check if a particular application needs to be re-written to prevent SQL injections.

Static Analysis Wassermann and Su [30] proposed an approach that uses a static analysis combined with automated reasoning. This technique verifies that the SQL queries generated in the application usually do not contain a tautology. This technique is effective only for SQL injections that insert a tautology in the SQL queries, but can not detect other types of SQL injections attacks.

JDBC Checker [8] statically checks the type correctness of dynamically generated SQL queries. JDBC Checker can detect SQL injection vulnerabilities caused by improper type checking of the user inputs. However, this technique would not catch more general forms of SQL injection attacks, because most of these attacks consist of syntactically correct and type-correct queries.

Dynamic Analysis Paros [23] is a free tool for testing for web application vulnerabilities without rewriting any scripts in the web application. This tool automatically scans for SQL injection vulnerabilities with pre-defined attack codes. Paros checks the contents of HTTP response messages to determine whether an SQL injection attack was successful or not. Paros is used in our evaluation in Section 5 for comparison with **Sania**.

Combined Static and Dynamic Analysis SQLCheck [28] checks SQL queries at runtime to see if they conform to a model of expected SQL queries. The model is expressed as a context-free grammar that only accepts legal queries. This approach uses a secret key to discover user inputs in the SQL queries. Thus, the security of the approach relies on attackers not being able to discover the key. Additionally, this approach requires the application developer to rewrite code to manually insert the secret keys into dynamically generated SQL queries.

AMNESIA [9] is a model-based technique that combines the static and dynamic analyses. In the static phase, AMNESIA uses a static analysis to build the models of the SQL queries that an application legally generates at each point of access to the database. In the dynamic phase, AMNESIA intercepts all the SQL queries before they are sent to the database and checks each query against the statically built models. Queries that violate the model are identified as SQL injection attacks. The accuracy of AMNESIA depends on that of the static analysis. Unfortunately, certain types of obfuscation codes and/or query generation techniques make this step less precise and results in both false positives and negatives.

SQLGuard [5] checks at runtime whether SQL queries conform to a model of the expected queries. The model is deduced at runtime by examining the structure of the query before and after a client's requests. SQLGuard requires the application developer to rewrite code to use a special intermediate library.

Machine Learning Approach Valeur *et al.* [29] proposed the use of an intrusion detection system (IDS) based on a machine learning technique. IDS is trained using a set of typical application queries, builds models of the typical queries, and then monitors the application at runtime to identify the queries that do not match the model. The overall IDS quality depends on the quality of the training set; a poor training set would result in a large number of false positives and negatives.

WAVES [12] is also based on a machine learning technique. WAVES is a web crawler that identifies vulnerable spots, and then builds attacks that target those spots based on a list of patterns and attack techniques. WAVES monitors the response from the application and uses a machine learning technique to improve the attack methodology. WAVES is better than traditional penetration testing, because it improves the attack methodology, but it cannot thoroughly check all the vulnerable spots like the traditional penetration testing.

Instruction-Set Randomization SQLrand [4] provides a framework that allows developers to create SQL queries using randomized keywords instead of the normal SQL keywords. A proxy between the web application and the database intercepts SQL queries and de-randomizes the keywords. The SQL keywords injected by an attacker would not have been constructed by the randomized keywords, and thus the injected commands would result in a syntactically incorrect query. Since SQLrand uses a secret key to modify keywords, its security relies on attackers not being able to discover this key. SQLrand requires the application developer to rewrite code.

Taint-Based Technique Pietraszek and Berghe [24] modified a PHP interpreter to track taint information at the character level. This technique uses a context-sensitive analysis to reject SQL queries if an untrusted input has been used to create certain types of SQL tokens. A common drawback of this approach is that they require modifications to the runtime environment, which diminishes the portability.

3 Design of Sania

In this section, we present our technique, **Sania**, which automatically tests for SQL injection vulnerabilities. **Sania** generates attack requests based on a syntactical analysis of the SQL queries generated by web applications. The novelty of **Sania** lies in that it exploits the syntactical knowledge of the SQL queries to generate attack requests.

3.1 Overview of Sania

An interactive web application ordinarily accesses its back-end database through a restricted private network. **Sania** is designed to be used by a web applications developer during the development and debugging phases, and thus is able to intercept SQL queries between an application and the database as well as HTTP requests between a client and the application. After capturing HTTP requests and SQL queries, **Sania** checks for any SQL injection vulnerabilities using the following three steps.

1. **Sania** identifies the *vulnerable spots* in the HTTP requests in which an attacker can embed maliciously crafted strings that cause SQL injection attacks. To identify the vulnerable spots, a web application developer sends innocent HTTP requests to the web application. Then, **Sania** captures the SQL queries generated from the HTTP requests, analyzes the syntax of the SQL queries, and identifies the vulnerable spots.

2. **Sania** generates attack requests that attempt to exploit the vulnerable spots where SQL injection attacks may occur. **Sania** can generate an elaborate attack because it leverages the knowledge obtained from the first step, that vulnerable spots appear in what context of the SQL queries. For example, **Sania** generates an attack request that targets two vulnerable spots at the same time; the special string injected in the former ignites the malicious code injected in the latter.

3. By sending the attack requests generated from the second step, **Sania** checks if SQL injection vulnerabilities lie in a web application. It uses the parse tree of SQL to check for vulnerabilities. If an attack request successfully injects malicious strings into an SQL query, the parse tree of the generated SQL differs from that generated in the first step when the innocent HTTP request is sent.

Since **Sania** generates attack requests based on the syntax of SQL queries, it can assess the safety of *every* spot that could potentially be vulnerable. An existing tool such as Paros [23] generates an attack code without knowledge of SQL queries. Therefore, Paros cannot generate an elaborate attack that targets two vulnerable spots at the same time. In addition, **Sania** discovers more vulnerabilities with less attack requests. This is because it suppresses the generation of meaningless attacks.

3.2 Specifying Vulnerable Spots

In SQL injection attacks, an attacker embeds maliciously crafted strings at certain points, called *vulnerable spots*, in HTTP requests whose values may appear in SQL queries. The vulnerable spots, in other words, have the potential to be security holes of SQL injection. The vulnerable spots appear in query strings, cookies, or other strings in HTTP requests.

Suppose that a HTTP request includes a query string such as "`id=555&cat=book`" and the generated SQL query is "`SELECT * FROM users WHERE user_id=555`". In this case, the query string has two sets of data separated by an ampersand (&), and the equality sign (=) divides each data set into two elements: *parameter* and *value*. Here, the parameters are `id` and `cat`, and the values are `555` and `book`. A parameter element is typically fixed, but an attacker can freely alter a value element. **Sania** detects every vulnerable spot by checking whether a value element appears in a leaf node of the parse trees of the generated SQL queries. In our example, "`555`" is a vulnerable spot because it appears in the SQL query, but "`book`" is not because it does not.

3.3 Crafting Attack Requests

Sania automatically generates attack requests by analyzing the parse tree of the SQL query generated from an innocent request. It embeds a maliciously crafted string, called an *attack code*, in the attack requests to exploit the vulnerable spots. It generates two types of attack requests: *linear* and *combination*. In a linear attack, **Sania** inserts an attack code to a single vulnerable spot. In a combination attack, it inserts attack codes in two or more vulnerable spots at the same time, both strings inserted in the vulnerable spots change the SQL syntax in concert.

3.3.1 Linear Attack

A vulnerable spot always appears in a leaf (i.e., terminal) node of a parse tree. The parent node of a leaf represents the non-terminal from which the terminal is derived. **Sania** generates an attack code according to type of the non-terminal to which a vulnerable spot belongs. In the example in Section 1, the non-terminals in the `name` and `password` fields both represent a *string expression* in the SQL grammar. An attack code in the string expression should have at least one single quote to escape a single quote in the SQL

query. If the non-terminal represents a *number expression*, an attack code should not contain a single quote.

In SQL injection, an attack code contains a meta-character to *end* the user input (usually a string) in a vulnerable spot. A vulnerable spot is divided into two parts with the insertion of a meta-character. The first part, called a `userInput`, contains a normal string that mimics an input from an ordinary user. The second part, called an `insertedSQL`, contains part of the SQL query that an attacker is trying to inject. Since the meta-character inserted in a vulnerable spot marks the end of a `userInput`, the strings in the `insertedSQL` are interpreted as SQL keywords.

Suppose that an attacker inserts the attack code "`yyy' or 1=1--`" into the `password` field in the Section 1 example. This attack code contains a single quote (meta-character) to end the user input and divide the `password` field into the `userInput` and `insertedSQL` parts. The `userInput` corresponds to the `yyy` and the `insertedSQL` corresponds to the "` or 1=1--`". The `userInput` contains a normal string `yyy` as if it were entered by the user. The `insertedSQL` contains a part of the finally injected SQL; the inserted meta-character activates the `insertedSQL` part.

Sania selects a meta-character, `userInput`, and `insertedSQL` based on the context in which a vulnerable spot would appear. **Sania** has a list of attack codes to test every non-terminal that has children terminals. To make a list of attack codes, we thoroughly investigated the SQL injection techniques in [1, 2, 5, 6, 13, 18, 19, 22, 25, 26]. We defined a total of 21 kinds of attack codes for 95 non-terminals. Since the list of attack codes is defined in XML, when a new attack code is developed a user can easily add it to the list.

In addition, **Sania** extensively uses the context in which a vulnerable spot appears to refine attack codes. To generate successful codes, the context in which a vulnerable spot appears must be more carefully examined, because it may be deeply set with nested parentheses. An attack code sometimes removes the right parentheses by including them in a quoted string or by commenting them out. Suppose that a web application generates the following SQL and **Sania** is attempting to inject a malicious string into the `name` field.

```
SELECT * FROM users WHERE
        (id=999 AND (name='xxx'))
```

Sania generates an attack code like "`' )) or 1=1--`" instead of "`' or 1=1--`" so that the syntax of SQL is not broken.

The list of attack codes in **Sania** is described as follows. Each attack code represented as a four-element tuple:

```
(userInput, metaCharacter,
        parentheses, insertedSQL).
```

A `metaCharacter` represents a meta-character that divides a vulnerable spot and a `parentheses` determines whether or not to insert parentheses. For each non-terminal, we described the possible combinations of a `userInput`, a `metaCharacter`, a `parentheses`, and an `insertedSQL`. For example, if a vulnerable spot belongs to a string expression, the attack code is (`λ | ϵ, ' | ", true, or '1'='1 | or "1"="1 | or 1=1-- | or 1=1;-- | or 1=1/*`). This means the userInput is either the input from the user (`λ`) or blank (`ϵ`), the metaCharacter is either the (`'`) or (`"`), the parentheses are used, and the insertedSQL is (`or '1'='1`), or (`or "1"="1`), or (`or 1=1--`), or (`or 1=1;--`), or (`or 1=1/*`). For an *item expression* in the `SELECT` statement, the attack code is (`λ, ϵ, false, from yyy-- | from yyy;-- | from yyy/*`). If "`SELECT id, xxx from users`" is issued by an application and `xxx` is a vulnerable spot, **Sania** inserts "`xxx from yyy--`" to the vulnerable spot, which results in "`SELECT id, xxx from yyy--from users`".

3.3.2 Combination Attack

A combination attack exploits two or more vulnerable spots at the same time to inject an SQL query. **Sania** currently tries to exploit two vulnerable spots at the same time. When two vulnerable spots, both of which belong to a string expression in the SQL grammar, are adjacent in the parse tree, **Sania** launches a combination attack. In the combination attack, it inserts a backslash to the former vulnerable spot to escape a single quote, which indicates the end of the string expression. Then, **Sania** inserts an SQL query to the latter vulnerable spot to cause an SQL injection.

Suppose that a web application issues the following SQL query: "`SELECT * FROM users WHERE name='`$\emptyset_1$`' and password='`$\emptyset_2$`'`" ($\emptyset_i$: vulnerable spot). In this example, **Sania** inserts a backslash to the former vulnerable spot ($\emptyset_1$) and a "` or 1=1--`" to the latter ($\emptyset_2$). If a backslash is not sanitized correctly, the resulting SQL becomes "`SELECT *FROM users WHERE name='\' and password=' or 1=1--'`". As a result, the `name` parameter is identified as "`' and password=`" because the injected backslash escapes the single quote. Thus, the `where` clause is evaluated to be true because "`1=1`" is always true and the following single quote is commented out by "`--`".

3.4 Validation

Sania compares the parse tree generated from an attack request with that generated from an innocent message to verify whether an attack was successful or not. If the parse

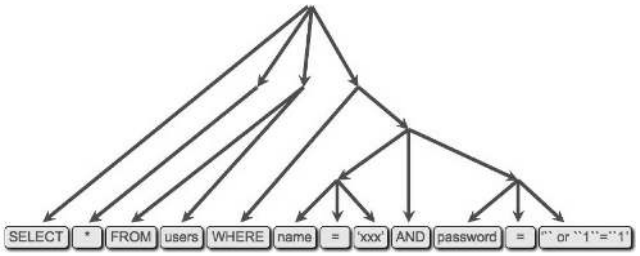

Figure 1. Properly sanitized parse tree

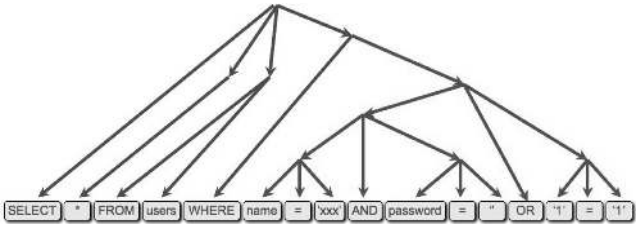

Figure 2. Improperly sanitized parse tree

tree generated from an attack request differs from that generated from an innocent request, **Sania** determines the attack was successful. Suppose that a web application issues the SQL query "`SELECT * FROM users WHERE name='ø'`" (ø: vulnerable spot), and the user input to vulnerable spot ø is "`' or '1'='1`". If the web application sanitizes the input properly, the parse tree will look like the one shown in Figure 1. If not properly sanitized, the parse tree will look like the one in Figure 2, where the structure of the parse tree is different from Figure 1.

Unfortunately, a web application generates SQL queries whose structures depend on the user inputs. Suppose that a web application generates the following SQL query: "`SELECT * FROM users WHERE id=ø`" (ø: vulnerable spot). If the web application allows an arbitrary arithmetic expression to be inputted in the `id` field, the SQL structure depends on the expression in the `id` field. If the `id` field contains a number, the corresponding node in the tree is a terminal node that represents a number. If the `id` field contains the addition of two numbers, the corresponding node has a structure that represents the addition of two numbers.

This type of SQL query is called a *dynamic query*. If a web application issues a dynamic query, **Sania** sometimes judges the application vulnerable to SQL injection, because an attack request that is different from that generated from an innocent request may result in the parse tree. This is a false positive if the application properly sanitizes all the inputs from the user. To avoid this problem, **Sania** allows the user to control the matching of parse trees; the user can specify what kind of subtree comes to a vulnerable spot

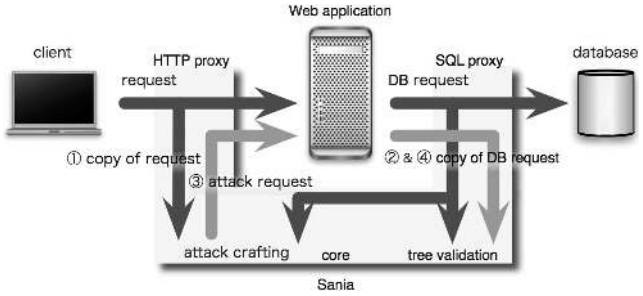

Figure 3. Packet capturing mechanism

if necessary. In **Sania**, the user can associate an attribute, called a *structure-attribute*, with a vulnerable spot. Table 1 lists the structure-attributes. For example, the user can associate an `arithmeticExpression` attribute with the `id` field in the above example. By doing this, the `id` field can contain an arbitrary arithmetic expression as well as a simple number.

4 Implementation

We implemented a version of **Sania** in Java that had 13,000 lines of code. In addition, it had a list of attack codes in XML that had 1,800 lines of code. We used JavaCC [16] and JSqlParser [17] to implement an SQL parser. As shown in Figure 3, **Sania** consists of an HTTP proxy, an SQL proxy, and a core component. The HTTP proxy captures the HTTP requests and responses, and the SQL proxy captures the SQL queries. The core component of **Sania** performs the tasks described in the previous section.

5 Experiments

This section presents our evaluation of **Sania** from the following points of view: efficiency and false positives in comparison with a public web application scanner.

5.1 Experimental Setup

To evaluate **Sania**, we selected six subjects. All are interactive web applications that accept HTTP requests from a client, generate SQL queries, and issue them to the underlying database. Table 2 lists the subject programs. Five of them (Bookstore, Portal, Event, Classifieds and Employee Directory) are free open source applications from GotoCode [7]. These were also used for evaluations in other papers [9, 28], and have been used to provide a service in the real world. Since GotoCode provides the applications in several languages, we used the JSP version for every application. The remaining one, E-learning, is a JSP and Java Servlet application provided by IX Knowledge Inc. [15]. It was

Table 1. Structure-attributes to widen allowable range of acceptable subelements for branch node

Name	Return value type	Acceptable expressions
arithmeticExpression	number	Number/mathematical statements
conditionalExpression	boolean	Conditional statement such as AND/OR statements
relationalExpression	boolean	Relational statement, which returns boolean value, but is not included in conditionalExpression such as LIKE/IS NULL statements
notExpression	boolean	A few statements, which can accept NOT expressions, such as BETWEEN, IN, and LIKE statements
subSelectExpression	result set	A few statements, which can accept sub-SELECT expressions, such as JOIN and FROM statements

Table 2. Subject programs used in our evaluation

Subject	Description	Language	LOC	Hotspots	
E-learning	Online Learning System	java(Servlet) & jsp	3682	14	(32)
Bookstore	Online Bookstore	jsp	11078	70	(116)
Portal	Portal for club	jsp	10051	96	(134)
Event	Event tracking system	jsp	4737	29	(50)
Classifieds	Online management system for classifieds	jsp	6540	35	(62)
Employee Directory	Online Employee Directory	jsp	3526	24	(38)

practically used in an intranet before, but it is no longer used after the newer version was released.

Table 2 shows each subject's name (Subject), a brief description (Description), the languages in which the application is written (Language), the number of lines of code (LOC), and the number of vulnerable spots (Hotspots) with the total number of injectable spots in parentheses.

Sania was used to try to discover the SQL injection vulnerabilities in these web applications. For comparison, we used Paros [23] to test these applications. Paros is the web application scanner that took second place out of the top 10 web vulnerability scanners at Insecure.Org [14]. At the top of the list was Nikto [21], but it is not designed to check for SQL injection vulnerabilities. Paros performs penetration testing that indiscriminately applies an attack code to every injectable spot. Paros determines that an attack is successful if the response message contains pre-defined strings that indicate database errors, such as "`JDBC.Driver.error`", because database errors indicate that the attack successfully changes and breaks the syntax of the SQL queries. Additionally, it determines that an attack is successful if the response after an attack differs from a normal response.

5.2 Results

Table 3 shows the number of resulting warnings reported by **Sania** and Paros. For each subject, the table reports the number of trials (trials), the number of raised warnings (total warns), the number of total vulnerabilities (vuls.), and the number of false positives (false positives). We checked for each warning whether it was truly vulnerable. This table reveals that **Sania** found more vulnerabilities for every subject and caused less false positives than Paros, using fewer trials.

Table 4. Details of vulnerabilities

Tool	Num.		Description
Sania	39	14	linear attacks (successful in E-learning)
		25	combination attacks
Paros	5	5	linear attacks (successful in E-learning)

5.2.1 Accuracy of Attacks

The details of the vulnerabilities are shown in Table 4. **Sania** and Paros found vulnerabilities in E-learning using linear attacks. **Sania** discovered 25 vulnerabilities in the GotoCode applications using combination attacks, but Paros could not because it does not support combination attacks. This result reveals that **Sania** can:

- Execute precise linear attacks. It could find more vulnerabilities (14 vuls.) than Paros (5 vuls.) even when they both used only linear attacks. This is because it generates an elaborate pinpoint attack according to the context in which a vulnerable spot appears.

- Execute powerful combination attacks. It found 25 vulnerabilities using combination attacks. A combination attack requires knowledge of the points where vulnerable spots appear in an SQL query. Therefore, it is hard for Paros to work out a combination attack.

Table 3. Results for Sania and Paros

Subject	**Sania**				Paros			
	trials	total warns	vuls.	false positives	trials	total warns	vuls.	false positives
E-learning	18	18	18	0	362	9	5	4
Bookstore	616	7	6	1	4802	8	0	8
Portal	831	16	7	9	5477	15	0	15
Event	250	4	4	0	1698	21	0	21
Classifieds	279	5	3	2	1210	6	0	6
Employee Directory	194	2	1	1	1924	13	0	13
total	2188	52	39	13	15473	72	5	67

Table 5. Details of false positives

Tool	Num.		Description
Sania	13	8	Length of attack code was too long
		3	Backslash mistakenly broke query
		1	Failed to delete inserted SQL query
		1	Authentication failed
Paros	67	16	Attack codes are mistakenly injected to state parameters
		15	Contents of response page was changed after editing
		13	These spots were already sanitized
		10	Length of attack code was too long
		9	Type of code does not match
		4	Duplicate warnings are received

5.2.2 False Positives

A false positive occurs when the test returns a positive result, but there is actually no fault (or error). Table 5 shows the details of the false positives. In total, **Sania** and Paros raised 13 and 67 false positives, respectively.

Length and Type Error If the length of an attack code is longer than that of its corresponding column defined in a database or the type of an attack code is improper, the database returns an error to its host web application. After catching this error message, the web application returns a response page to handle the error. If the response page urges the web application to issue unintended SQL queries, **Sania** obtains them and mistakenly raised 8 warnings. Paros found the unintended contents of the response page, and caused 10 false positives for the length problem and 9 false positives for the type problem.

Misallocated Backslash A backslash in an attack code escapes a single quote in the SQL grammar. **Sania** caused 3 false positives when it mistakenly inserted a backslash into inappropriate vulnerable spots. For example, a page of `Portal` issues an SQL query, which has a vulnerable spot $ø_1$ and subsequently issues another SQL query that has two vulnerable spots $ø_1$ and $ø_2$. **Sania** embeds a backslash to the vulnerable spot $ø_1$ intending to execute a combination attack, and breaks the first SQL query. This problem can be avoided by more carefully annotating the vulnerable spots.

Database Record Conflict The data inserted into the database should be deleted before the next trial starts, because it would adversely affect the results of the following trials. In penetration testing, if a tool has no function to delete inserted data items, the later trials are always evaluated to be safe. In our evaluation, a page of `Classifies` initially checks whether a name specified in an HTTP request already exists. If the name does not exist in the database, it inserts the name. Otherwise, it returns an error page. **Sania** caused one false positive when it failed to delete the inserted data items.

Field Data Conflict **Sania**'s remaining false positive was caused when an attack code was used in the password confirmation field. This field is used to confirm that the two passwords inputted by the user do match. The subject, Bookstore, has a password confirmation field, "`member_password2`", to verify that the entered password matches "`member_password`" field. **Sania** inserts different values to these fields and raises a false alarm because it does not know that these field should have the same string.

Attacking Potentially Safe Spots Some query-strings, called *state parameters*, are potentially safe because they never become a part of the SQL queries. A state parameter controls the page transition to handle the client's requests dynamically. A query-string, "`FormAction=insert`", is used as a state parameter in `Portal` in order to insert new article information. If an attack code is applied to this state parameter, the web application makes the client go back to the original page with an error, "`java.sql.SQLException:`

`Can not issue empty query`", without handling any SQL queries from the requested page. Paros recognizes this as a successful attack, resulting in 16 false positives. The additional 13 false positives in Paros were caused by attacking already-sanitized spots. **Sania** avoids these false positives by properly selecting vulnerable spots before starting the test.

Mishandling of Dynamic Contents Some web applications return a dynamically generated page that contains the values entered by the user. Paros always misjudges this type of web applications as being vulnerable and caused 15 false positives. This is because the contents of the response page changes if the user input changes.

Duplicate Warnings The four remaining Paros' false positives are duplicate warnings. They are already classified into the other categories above.

5.3 Testing a Real Product

After **Sania** was proven effective in our experiments, we had a chance to test a production-quality web application developed by IX Knowledge Inc. [15] on March 28, 2007. This application, RSS-Dripper, provides RSS information to the user based on their previous choices. RSS-Dripper is written in Java Servlet and JSP, developed on Struts [27], and was in the final step of the development process just before being shipped when we tested it.

Sania detected one SQL injection vulnerability after testing 33 attack requests. RSS-Dripper was vulnerable to a combination attack. We analyzed the source code of RSS-Dripper to find and modify the security hole. This test shows there are SQL injection vulnerabilities in production-quality web applications that have been recently developed, and **Sania** is effective in these real applications.

6 Conclusion

We presented our new web application security technique, **Sania**, which is designed to check for SQL injection vulnerabilities in the development and debugging phases. **Sania** intercepts SQL queries and automatically generates elaborate attacks based on the syntax of potentially vulnerable spots in the SQL queries. **Sania** assesses the safety of these spots by comparing the parse trees of the intended SQL query and that resulting after the attack. By analyzing the syntax in the parse tree of SQL queries, it is possible to generate precise pinpoint attack requests. **Sania** has been proved effective; it found 39 SQL injection vulnerabilities and incurred only 13 false positives. Paros, a popular scanner for web application vulnerabilities found 5 vulnerabilities and caused 67 false positives.

References

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

[2] C. Anley. (more) Advanced SQL Injection. *White Paper, Next Generation Security Software Ltd.*, 2002.

[3] Armorize Technologies. http://www.armorize.com/.

[4] S. Boyd and A. Keromytis. SQLrand: Preventing SQL injection attacks. *In Proceedings of the Applied Cryptography and Network Security (ACNS)*, pages 292–304, 2004.

[5] G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using parse tree validation to prevent SQL injection attacks. *In Proceedings of the 5th International Workshop on Software Engineering and Middleware SEM*, pages 106–113, 2005.

[6] Ferruh.Mavituna. SQL Injection Cheat Sheet. http://ferruh.mavituna.com/makale/sql-injection-cheatsheet/.

[7] GotoCode. http://www.gotocode.com/.

[8] C. Gould, Z. Su, and P. Devanbu. JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications. *In Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pages 697–698, 2004.

[9] W. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. *In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 174–183, 2005.

[10] W. Halfond, J. Viegas, and A. Orso. A Classification of SQL-Injection Attacks and Countermeasures. *Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE)*, 2006.

[11] Hibernate. hibernate.org. http://www.hibernate.org/.

[12] Y. Huang, S. Huang, T. Lin, and C. Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In *Proceedings of the 12th International World Wide Web Conference (WWW03)*, pages 148–159, 2003.

[13] iMPERVA. Blind SQL Injection. http://www.imperva.com/application_defense_center/white_papers/blind_sql_server_injection.html.

[14] Insecure.Org. http://insecure.org/.

[15] IX Knowledge Inc. http://www.ikic.co.jp/.

[16] JavaCC. https://javacc.dev.java.net/.

[17] JSqlParser. http://jsqlparser.sourceforge.net/.

[18] C. A. Mackay. SQL Injection Attacks and Some Tips on How to Prevent Them. http://www.codeproject.com/cs/database/SqlInjectionAttacks.asp.

[19] S. McDonald. SQL Injection Walkthrough. http://www.securiteam.com/securityreviews/5DP0N1P76E.html, May 2002.

[20] MITRE. Common Vulnerabilities and Exposures (CVE). http://cve.mitre.org/.

[21] Nikto. CIRT.net. http://www.cirt.net/code/nikto.shtml.

[22] OWASP. Testing for SQL Injection. http://www.owasp.org/index.php/Testing_for_SQL_Injection.

[23] Paros. Parosproxy.org. http://www.parosproxy.org/.

[24] T. Pietraszek and C. Vanden Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. *In Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 124–145, 2005.

[25] K. Spett. SQL Injection: Are your web applications vulnerable? *SPI Labs White Paper*, 2004.

[26] Steve Friedl's Unixwiz.net Tech Tips. SQL Injection Attacks by Example. http://www.unixwiz.net/techtips/sql-injection.html.

[27] Struts. Apache Struts project. http://struts.apache.org/.

[28] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. *Annual Symposium on Principles of Programming Languages (POPL)*, pages 372–382, 2006.

[29] F. Valeur, D. Mutz, and G. Vigna. A Learning-Based Approach to the Detection of SQL Attacks. *In Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, pages 123–140, 2005.

[30] 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)*, pages 70–78, 2004.