Toward A Taxonomy of Techniques to Detect Cross-site Scripting and SQL Injection Vulnerabilities

Yonghee SHIN, Laurie WILLIAMS, *Members, IEEE*

***Abstract*—Since 2002, over half of reported cyber vulnerabilities are caused by input validation vulnerabilities . Over 50 % of input validation vulnerabilities were cross-site scripting and SQL injection vulnerabilities in 2006, based on the (US) National Vulnerability Database. Techniques to mitigate cross-site scripting and SQL injection vulnerabilities have been proposed. However, applying those techniques without precise understanding can result in a false sense of security. Clearly understanding the advantages and disadvantages of each security technique can provide a basis for comparison of those techniques. This survey provides a taxonomy of techniques to detect cross-site scripting and SQL injection vulnerabilities based upon of 21 papers published in the IEEE and ACM databases. Our taxonomy characterizes the detection methods and evaluation criteria of the techniques. The taxonomy provides a foundation for comparison among techniques to detect cross-site scripting and SQL injection vulnerabilities. Organizations can use the comparison results to choose appropriate techniques depending on available resources.**

***Index Terms*-- Software Engineering, Software/Program Verification, Software Quality, Security and Privacy Protection**

1. INTRODUCTION

A vulnerability is a weakness in a system caused by a flaw in design, a coding error, or incorrect configuration such that execution of a program can violate the implicit or explicit security policy [11, 21]. Input validation vulnerabilities (IVVs) are caused by lack of checking for invalid input. Input can be validated against predefined or known invalid input characters or string patterns. For example, many web sites limit the maximum length for login ID and generate an error when the input length exceeds the required maximum length. An attack via an IVV occurs when an attacker takes advantage of an IVV to make use of an asset in a system for the attacker's purpose [11]. For example, an attacker may enter code into an input field in such a way that the code is executed on a client's machine or at the server. Common IVVs are cross-site

Manuscript received December 5, 2007. This work was supported by the National Science Foundation CAREER Grant No. 0346903 and CyberTrust Grant No. 0716176.

Yonghee Shin is with North Carolina State University, Raleigh, NC 27695 USA (e-mail: yonghee.shin@ncsu.edu).

scripting (XSS), SQL injection, and buffer overflow vulnerabilities. Figure 1 shows the number of IVVs reported to National Vulnerability Database (NVD) [1] since 1995. The quantity of IVVs that has been reported has dramatically increased since 2004 and more than half of IVVs are XSS, SQL injection, and buffer overflow vulnerabilities. The increase of *reported* vulnerabilities can be interpreted in two ways: (1) the increase in the number of actual vulnerabilities and (2) the increase in concern about vulnerabilities when the number of actual vulnerabilities was not increased (causing an increase in reporting). The concern about vulnerabilities comes from both attackers and normal users. Therefore, the increase in vulnerability report indicates we need to make more efforts to reduce IVVs.

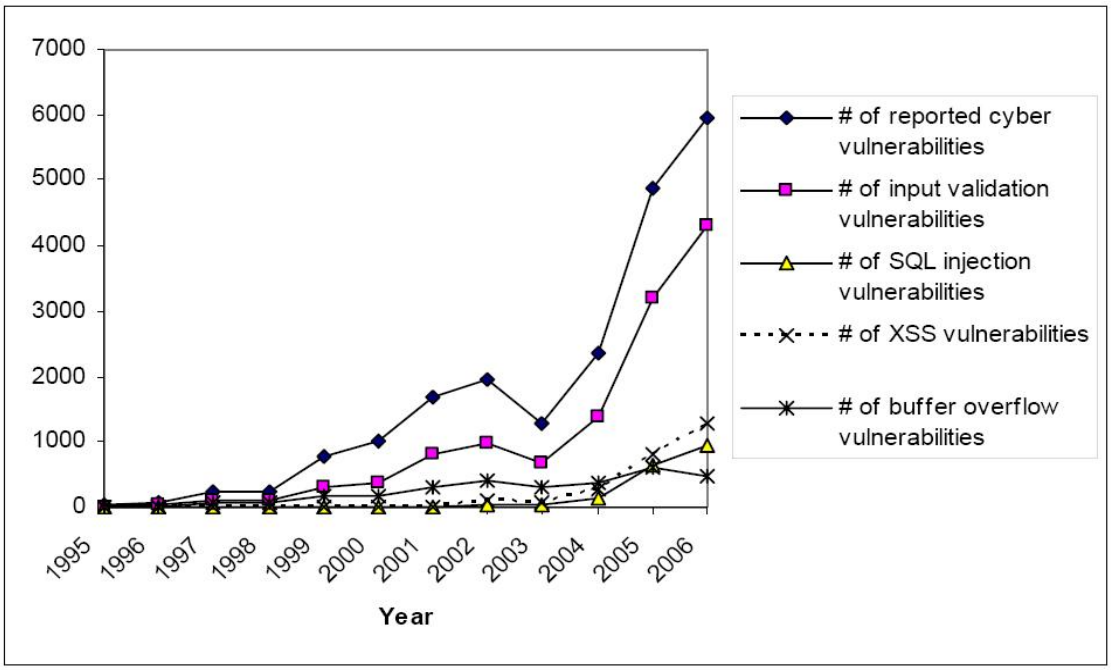

Fig 1. The number of reported cyber vulnerabilities

A diverse set of techniques have been introduced to detect *XSS and SQL Injection Vulnerabilities (XSIVs)* [2, 5, 6, 8-10, 12-16, 18, 23, 24, 27-33]. Each technique has advantages and disadvantages and clear understanding of these advantages and disadvantages can provide a

Laurie Williams is with North Carolina State University, Raleigh, NC 27695 USA (e-mail: williams@csc.ncsu.edu).

basis for comparison between detection techniques. Organizations can compare the vulnerability detection techniques and choose appropriate techniques for the organizations' unique operational environments and available resources.

The objective of this research is to analyze and compare XSIV detection techniques and to identify areas for future research directions in vulnerability detection techniques. We categorize the XSIV detection techniques and provide the definition of each category. Although classifications and comparisons of XSIV detection techniques have been provided in the literature we reviewed for this research, those classifications and comparisons are mostly as a form of related work to explain the context of a XSIV detection technique, and none of them are as comprehensive as our taxonomy.

Note that comparing techniques based on information in the papers can be difficult because papers described their approaches with differing levels of detail. Therefore, our survey can be used as a guide to compare techniques and to find future research directions rather than to comprehensively judge the quality of known techniques.

We limited the scope of this paper to XSIVs and did not include the techniques for buffer overflow vulnerabilities to focus on comparatively recent and fast growing vulnerabilities. Figure 2 shows the percentage of XSS, SQL injection and buffer overflow vulnerabilities among total IVVs. The rate of reported XSIVs among IVVs has increased since 2001. However, the rate of reported buffer overflow vulnerabilities has decreased since 2003.

The rest of this paper is organized as follows. Section 2 provides background on XSS and SQL injection vulnerabilities. Section 3 explains the overall structure of our taxonomy. Sections 4 and Section 5 describe the two dimensions of our taxonomy, detection methods and evaluation criteria, in detail. Section 6 concludes and discusses future work. Appendix A summarizes our

taxonomy.

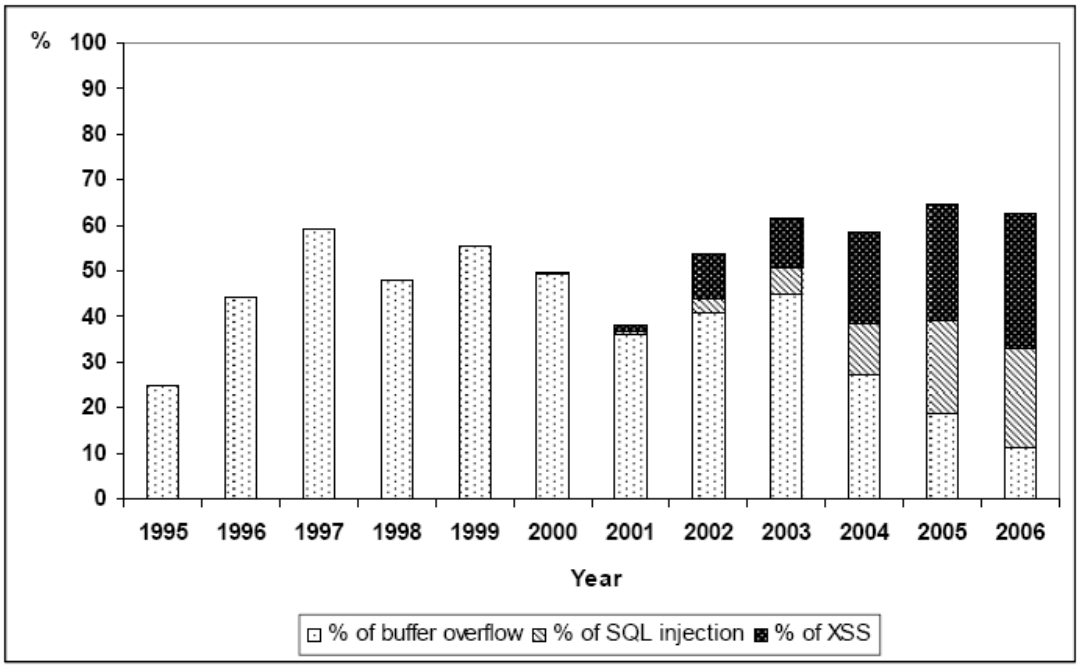

Fig 2. The percentage of XSS, SQL injection, and buffer overflow vulnerabilities among IVVs

2. BACKGROUND

This section describes XSS and SQL injection vulnerabilities in detail and provides representative examples of each.

2.1 Cross-site Scripting

Cross-site scripting (XSS) vulnerabilities allow attackers to insert malicious scripts as a part of user input and the script is executed at other user's browser due to the lack of input validation. We provide a simple example of attacks exploiting XSS vulnerabilities. Consider a search engine that returns the search results including the same query given by a user. If the user input includes a script and the returned result page does not encode the script into HTML code, the script in the returned result page will be executed. For example, assume a user entered an executable script as a query for a search engine as in Figure 3.

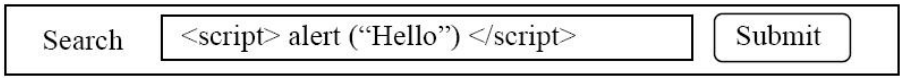

Fig 3. Script as a user input

If the search engine returns the result with the same query, which is a script in this case, the browser will execute the script and display an alert with a “Hello” message as in Figure 4.

Fig 4. Result from Figure 3

XSS attacks also can be used to access user’s critical information by stealing HTTP cookies [20]. When a web application needs to keep track of a communication channel between a web server and a user’s browser, a web server sends a HTTP cookie that includes the information that the server can identify the user. If an attacker can access the cookie at his or her browser, the attacker can disguise as the user and can access critical information such as credit card number that only the user was supposed to access. Web sites on which users can post messages to a message board shared with other users and that require users to log in to the system to use the message board are potentially vulnerable to this XSS attack. A procedure of cookie stealing is as follows in this case:

Step 1. An attacker finds a web site on which users can post messages to a message board shared with other users and that requires users to log in to the system to use the message board.

Step 2. The attacker posts the script in Figure 5 that sends cookie information to the attacker’s web site when the script is executed.

```
<script> document.location=
'http://www.attacker.com/cgi-bin/getcookie.cgi?'
+ document.cookie </script>
```

Fig 5. A malicious script for XSS attack

Step 3. A user logs in to the web site and reads the message that the attacker has posted.

Step 4. The script in Figure 5 is executed and the cookie information of the user is sent to the attacker's site.

Step 5. The attacker can access the user's critical information using the cookie.

2.2 SQL Injection

SQL injection vulnerabilities allow attackers to insert SQL commands as a part of user input. When an SQL query is constructed dynamically with maliciously-devised user input containing SQL keywords, attackers can gain access or modify critical information such as a credit card number in a database without proper authorization. For example, Figure 6 shows a sample program in Java using JDBC that uses a SQL query to authenticate a user via `id` and `password`. The query is dynamically created via the program statement in bold. In the query in Figure 6, `id` and `password` are obtained

```
public boolean isRegistered(String id,
                            String password) {
 // Connect to a database
 …
 Connection dbConn =
       DriverManager.getConnection(to);

 // Create a query
 String sqlQuery =
      "SELECT userinfo FROM users
       WHERE id = '" + id + "'
       AND password = '" + password + "'";
 Statement stmt =
       dbConn.createStatement();

 // Execute a query
 ResultSet rs =
       stmt.executeQuery(sqlQuery);
 if(rs != null)
        return true;
 else
        return false;
}
```

Fig 6. An example of SQL query with a vulnerability

via user input.

In the previous example, a normal user input would look like Figure 7, though the password appears rather than the typical "******".

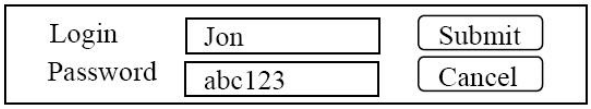

Fig 7. Web form with normal user input

However, an attacker can enter the input for the values of login ID and password through a web form as in Figure 8.

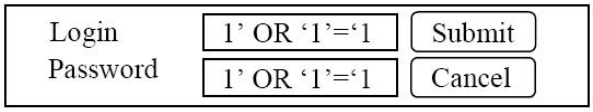

Fig 8. Web form with malicious user input

Which would generate the following query:

```
SELECT userinfo FROM users
WHERE id = '1' OR '1' = '1'
AND password = '1' OR '1' = '1';
```

Because the given input makes the WHERE clause in the SQL statement always true (a tautology), the database returns all of the user information in the table. Therefore, the malicious user has been authenticated without a valid login ID and password.

The use of tautology is a well-known SQL injection attack [3, 8, 31]. However, there are other types of SQL injection attacks using multiple SQL statements or stored procedures. SQL clauses such as "UNION SELECT", "ORDER BY", and "HAVING" are sometimes used to gain knowledge about database structure that can be used for SQL injection attacks. The attackers also can gain knowledge about database structure by exploiting error messages from an SQL command failure [3, 26] or simply by trial and error [25].

Many languages and libraries for the languages such as Java, C++ and PHP provide a way to

prevent SQL injection attacks by using prepared statements. A prepared statement escapes special characters in user input when the user input is given as a binding variable for the prepared statement. Escaping is usually performed by adding a backslash in front of a special character so that the character is interpreted as a normal character instead of being interpreted in a special meaning. Figure 9 shows a use of prepared statement for the example given in Figure 6. The variable `id` is bound to the first binding variable and the variable `password` is bound to the second binding variable in the `sqlQuery` in Figure 9.

Unfortunately, legacy code and applications written by novice programmers often do not use prepared statements. Furthermore, prepared statements are not inherently safe; one can misuse a prepared statement by using string concatenation instead of using a binding variable due to a poor programming practice. Additionally, prepared statements also do not allow parameterized table names and column names. If programmers want to parameterize table names and column names, programmers must use normal SQL statements instead of prepared statements [19].

```
public boolean isRegistered(String id,
                            String password) {
 // Connect to a database
 …
 Connection dbConn =
       DriverManager.getConnection(to);

 // Create a query
 String sqlQuery =
      "SELECT userinfo FROM users
       WHERE id = ?
       AND password = ?";
 Statement stmt =
       dbConn.prepareStatement(sqlQuery);
        stmt.setString(1, id);
        stmt.setString(2, password);

 // Execute a query
 ResultSet rs =
       stmt.executeQuery();
 if(rs != null)
        return true;
 else
        return false;
```

Fig 9. Using a prepared statement to prevent an SQL injection vulnerability

3. OVERVIEW OF TAXONOMY

XSIV detection techniques should be able to identify vulnerabilities accurately during development or operation in the field. XSIV detection techniques requiring higher accuracy and less effort are more likely to be adopted by an organization. We created a taxonomy for XSIV detection techniques to help the understanding of accuracy of XSIV techniques and their evaluation criteria including ease-of-use for fair comparison. Our taxonomy consists of two dimensions: detection methods and evaluation criteria. Each dimension consists of sub-dimensions that represent different aspects of the dimension. Each sub-dimension consists of categories that classify the techniques according to the corresponding aspects. Figure 10 shows the hierarchy of our taxonomy. The numbers beside the name of sub-dimensions are the sections of this paper in which the sub-dimension is explained in detail.

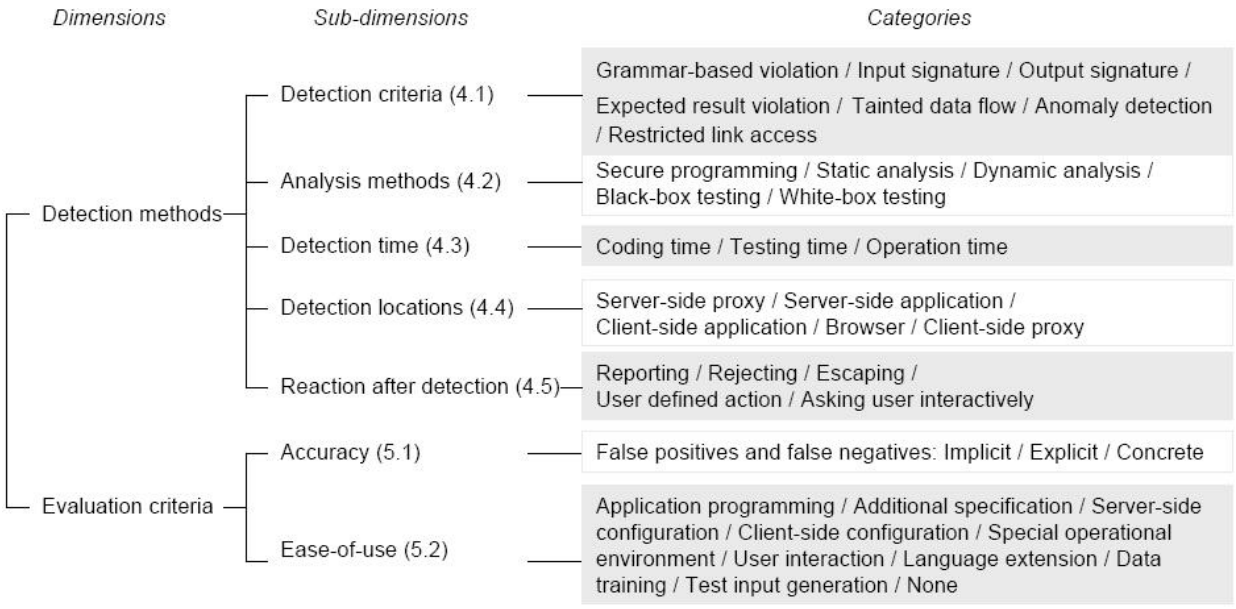

Fig 10. Hierarchy of taxonomy

We reviewed 23 techniques in 21 papers from the ACM[1] and IEEE[2] digital libraries for our survey. We collected the papers by searching the keywords "SQL injection", "cross-site script" and "XSS" from those libraries. During our search, the IEEE digital library contained seven papers on XSS and eight papers on SQL injection. Similarly, the ACM digital library contained 35 papers on XSS and 50 papers on SQL injection. From these 100 papers, we selected 21 papers closely related to XSIV detection techniques. Though the papers we surveyed are not exhaustive, the techniques used in the 21 papers are representative of the current techniques that we analyzed. Table 1 shows the summary of the literature we reviewed.

Table 1. Summary of literature for XSIV detection techniques

Digital libraries	**Papers selected**	**Techniques reviewed**
IEEE	8	8
ACM	13	15
Total	21	23

[1] http://www.acm.org/
[2] http://www.ieee.org/

Huang et al. [12] and Kals et al. [16] provided techniques for both XSS and SQL injection. However, each of those techniques is classified into a different category in our taxonomy because the detection methods for XSS and SQL injection were different. Therefore, we counted the techniques described in [12] and [16] separately. We used the names of techniques described in the reviewed papers, such as SecuBat [16] and WAVES [12]. When no name was given in a paper, we created a name for consistent and easy reference, for example, DetectCollectXSS^{n} [14]. The 'n' superscript after the name of a technique indicates a name that we created.

Table 2 shows the papers we surveyed for each vulnerability type. When a paper provides their implementation for only one vulnerability type, but the approach could be used for both types, we categorized the technique based on the approach described in the paper rather than the implemented tool specified in the paper. For example, Su and Wasserman claim their technique can detect both SQL injection and XSS as well as other IVVs [31]. However, they introduced a tool to detect only SQL injection vulnerabilities in their paper [31]. In this case, we categorized [31] as a technique to detect both SQL injection and XSS vulnerabilities.

Table 2. Literature according to vulnerability types

Vulnerability types	Literature
XSS-only	[10], [12], [14], [16], [18], [24], [29]
SQL-injection-only	[2], [6], [8], [9], [12], [16], [28], [33]
Both	[5, 17][3], [13], [15],[22, 32][3], [23], [27], [30], [31]

4. DETECTION METHODS

Vulnerability detection methods can be classified according to the meaning of detection, how, when, and where XSIVs are analyzed and detected, and how the results of detection are handled. Knowing

[3] The approaches presented in [5] and [17] are very similar even though their implementation detects different vulnerabilities. Additionally one of the authors co-authored both of the papers. Therefore, we treated them as a single technique that can detect various

the detection time in the software life cycle, detection location in the system, and the reaction after the vulnerability detection as well as detection methods is useful to determine the detection technique(s) to be used for a particular development or operational environment.

4.1 Detection criteria

Vulnerability detection techniques have their own criteria to decide the existence of vulnerabilities. For example, one technique determines that an application is vulnerable when there is an input flow from external source to a certain statement in an application, while another technique determines that an application is vulnerable when user input includes certain characters. We identified seven categories of detection criteria: grammar-based violation, input signature, output signature, expected result violation, tainted data flow, anomaly, and restricted link access detection. Each of these categories will be described, and examples of the categories from our literature search will be provided.

4.1.1 Grammar-based violation detection

Grammar-based violation detection uses the grammatical structure of SQL commands or script languages to detect vulnerabilities. XSIVs can happen when an external input includes a part or whole statement that can be interpreted and executed at runtime. The idea of grammar-based violation detection techniques is to construct two grammatical representations of SQL statements or script languages such as finite state machines (FSM) or parse trees for the statement that the external input is used, one with and one without user input. If the two grammatical representations are not the same, grammar-based violation detection techniques consider the user input to be including a malicious command that changes the intended behavior of the statement. For example, Fig 11 shows an FSM representation for the example shown in Figure 6 with only one variable in the WHERE clause [8, 33].

---

types of vulnerabilities even though [5] is not included in the 21 papers we found from ACM and IEEE digital library. [22] and [32] are also the same case. Only [22] is included in the 21 papers we found from ACM and IEEE digital library.

An SQL injection vulnerability is detected because the FSM constructed from actual user input (`1' OR '1' = '1`) has different grammatical representation from the one constructed without user input.

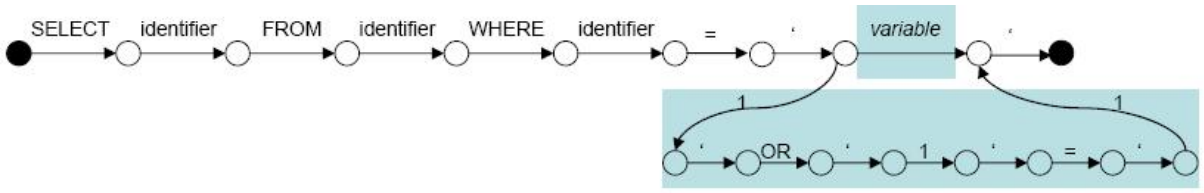

Fig11. Finite state machine for a SQL query[4]

SQLStoredProcedure[n] [33], AMNESIA [8], and SQLGuard [6] compare automata or parse trees for an SQL query constructed with and without user input, and detect differences between them. SQLCheck [31] detects invalid structure in the parse tree constructed with user input. The parse tree is constructed in a way that any SQL keywords contained in a user's input create a non-unique root in the parse tree. Therefore, the existence of non-unique roots in the parse tree indicates the user's input is malicious. In the literature we reviewed, all of the grammar-based techniques detected only SQL injection vulnerabilities except SQLCheck, which is also able to detect XSS and other types of injection vulnerabilities. [31].

4.1.2 Input signature detection

Input signature detection detects special characters or keywords from user input by scanning external input to seek *blacklist*. Blacklist is a set of special characters or keywords that are known to be used to create malicious input. The single quotation mark(') is often blacklisted for SQL injection. Left and right bracket ("<" and ">") are often blacklisted for XSS.

If user input includes SQL keywords, as in Figure 8, and the input is used to construct an SQL query without any validation or modification, a technique based on input signature detection considers the application vulnerable. Vulnerabilities can be avoided by changing malicious input to non-malicious

input or by rejecting the malicious input. For example, escaping (adding a backslash) in front of a single quotation mark in an SQL query makes the single quotation mark interpreted as a part of character constant instead of being interpreted as a part of SQL keyword. The SQL query below is the result of escaping user input in Figure 8.

```
SELECT userinfo FROM users
WHERE id = '1\' OR \'1\' = \'1'
AND password = '1\' OR \'1\' = \'1';
```

SQL DOM [28], AntiMaliciousInjection[n] [23], and DetectCollectXSS[n] [14] escape malicious characters. Reaction after detection is described in Section 4.5 more in detail. When the blacklist is too restrictive, the program could sacrifice usability. When the blacklist is not restrictive enough, vulnerabilities can be exploited. Therefore, the input signature must be carefully determined so that an application performs the functionality correctly while still detecting malicious input.

4.1.3 Output signature detection

Output signature detection detects special characters, keywords, or certain statements from the output of application execution to find the evidence that an application is vulnerable. For example, in the sample application in Figure 6, if user input for login ID and password are " ' ", the resulting SQL query will have a quotation mark that results in an non-matching quotation mark as below.

```
SELECT userinfo FROM users
WHERE id = '''
AND password = ''';
```

This query will cause an SQL syntax error reported by the database management system. The SQL syntax error reveals that the application does not validate user input properly and allows malicious user input reaches a *target statement*. A target statement is a statement in the code that can be exploited by

[4] Figure 11 was adapted from the figure in [8] for our example.

attackers when malicious input is given.

Some of automated black-box testing techniques such as SecuBat [16] and WAVES [12] use output signature detection. DetectCollectXSS[n] [14] provides both input and output signature detection modes. However, the detection ability of output signature detection depends on the completeness of the list of output signature for an application.

4.1.4 Expected result violation detection

Expected result violation detects the mismatch between the execution output and expected results described by a predefined specification. Expected result violation differs from techniques for output signature in that output signature detects predefined output patterns such as special keywords or characters and applies the same output patterns to every execution result, while expected result violation detects a specific value that a developer expects as a result of an instance of execution. SecuBat [16] finds a executable script that was entered as input for a web request from the HTTP response. If the input script is found from the response web page without any modification, the server did not perform any input validation and the script can be executed at the user's browser. Therefore the web page is vulnerable to XSS attacks. XSSTestGen[n] [24] stores predefined test cases to detect XSS vulnerabilities and compares the results with expected output.

4.1.5 Tainted data flow detection

Tainted data flow detection analyzes external input data flow and detects whether the input is used for a target statement. External input includes input from user interface, network interface or any other interface whose input comes from an external source.

Pixy [15] statically checks if user input can reach a target statement without being processed by an input validation routine by performing data flow analysis. PQLMatcher[n] [27] identifies an object flow that matches the specification written in PQL (Program Query Language) by developers and performs

a user-defined action when the match was found at runtime. An example of user-defined action is to escape special characters in a string object. XSSTestGen^n [24] statically analyzes the control flow of web applications to trace user input to an output node and generates test cases based on the result of the analysis. A web page is potentially vulnerable if user input reaches an output node and vulnerable if user input reaches an output node without being redefined. If user input does not reach an output node, the web page is not vulnerable. XSSTestGen^n generates test cases only for the web pages that are vulnerable or potentially vulnerable. WASP [9] combined both dynamic tainted data flow analysis and grammar-based violation detection. Some tainted data flow detection techniques return false warnings when malicious input cannot reach a target statement because the input has been validated before the input reaches a target statement. Some techniques ([13, 24, 27]) insert input validation code to the potentially vulnerable location in the code identified by static tainted data flow analysis to detect actual vulnerabilities and prevent false warnings at runtime.

4.1.6 Anomaly detection

Anomaly detection techniques generate warnings when the runtime behavior of an application deviates from the normal behavior that was recorded during the training period. Only normal actions are performed on the application without any abnormal behavior that could be considered as a security attack during training period. The technique then compares runtime behavior to pre-recorded normal behavior. However, anomaly detection techniques cannot detect attacks that are not modeled during training period.

WAVES [12] records the behavior of a web browser such as directories accessed or libraries loaded when dynamic content such as JavaScripts or ActiveX controls is executed in training period. If runtime behavior of a script is different from the recorded normal behavior, WAVES detects XSS. AnomalyDetection^n [22, 32] creates models about the characteristics of normal user input based on

statistical information that can be obtained from non-malicious user input, for example, user input length and character distribution. If runtime behavior deviates from the model, AnomalyDetection^n detects XSS [22] or SQL injection [32]. JavaScriptMozilla^n [10] uses security policy that describes the normal behavior of JavaScripts. If runtime behavior violates the policy, JavaScriptMozilla^n detects XSS.

4.1.7 Restricted link access detection

Restricted link access techniques detect an access to an external web site with potential XSS attacks at runtime when a user clicks a link to an external web site that is not in the same domain of the current web site. Noxes [18] raises an alert when the user tries to access a web link to an external web site. Users need to determine interactively if a link is actually vulnerable or not. However, users can also define filter rules to reduce their interaction.

4.1.8 Comparison of detection criteria

As we have seen in the previous subsections, the accuracy of detection varies depending on techniques. Expected result violation detection can detect vulnerabilities most accurately because expected result violation is based on the assumption that a test already knows the possible malicious input and the behavior of the system for the specific input. Therefore, the tester must know about how to attack the vulnerabilities and prepare the expected results for each specific application. On the contrary, other detection criteria do not require application-specific knowledge. Grammar-based violation detection can detect as accurately as expected result violation detection in the best case. However, the accuracy of detection varies depending on algorithms and analysis methods which will be discussed in Section 4.2. Dynamic analysis methods can usually detect vulnerabilities more precisely than static analysis methods for the same detection criteria because dynamic analysis methods do not use the approximation that static analysis methods use due to the complexity of code analysis.

Input and output signature detection techniques can be less accurate than expected result violation and grammar-based violation detection because their detection ability depends on the completeness of signatures identified and vulnerable signatures vary depending on context. For example, an application can allow a single quotation mark in a user name as a part of valid user input, while another application considers every single quotation mark vulnerable. However, input and output signature detection is simple to implement compared to expected result violation and grammar-based violation.

Tainted data flow techniques detect only the flow of suspicious information rather than detecting actual malicious input. Therefore, tainted data flow can generate false alerts per se. Tainted data flow detection can maximize effectiveness of vulnerability detection when it is combined with other detection criteria such as input signature detection. Anomaly detection relies on a statistical model constructed during training period with non-malicious input and does not detect actual malicious input, either. However, both of tainted data flow and anomaly detection are good for detecting previously-unknown vulnerabilities. The accuracy of restricted link access detection depends on how the restricted link is determined. We found only one restricted link access technique [18] which detects access to external web sites and access to web sites that users designate.

4.1.9 Summary of literature

Table 3 shows the classification of techniques according to detection criteria.

Table 3. Literature according to detection criteria

Detection Criteria	XSS	SQL injection	Both
Grammar-based violation		[6], [8], [9][5], [33]	[5, 17], [31]
Input signature	[14], [29]	[2], [28]	[23], [30]
Output signature	[14]	[12], [16]	
Tainted data flow	[24]	[9]	[13], [15], [27]
Anomaly detection	[10], [12]		[22, 32]
Expected result violation	[16], [24]		

[5] Some techniques use combined techniques. In Table 3 WASP [9] is counted twice, for grammar-based violation and for tainted data flow. DetectCollectXSS[n] [14] is also counted twice, for input signature detection and for output signature detection. XSSTestGen[n] [24] is counted twice for tainted data flow and expected result violation.

Restricted link access	[18]		

4.2 Analysis methods

XSIV detection techniques use various analysis methods to detect vulnerabilities. Analysis methods consist of five categories: secure programming, static analysis, dynamic analysis, black-box testing and white-box testing.

4.2.1 Secure programming

Secure programming is an approach to reduce security vulnerabilities by implementing user input validation routines in the application source code or by using existing routines for user input validation provided by vendors or standard libraries for languages. SQL DOM [28] provides a set of classes automatically generated from existing database schema. These classes allow programmers to create a SQL query using predefined class methods instead of using dynamic concatenation of strings. The constructors of these classes escape special characters to prevent SQL injection attacks. AntiMaliciousInjection[n] [23] automatically injects input validation routines in the server side scripts that process user input. The drawback of secure programming is that developers need to be trained to write secure code or to learn how to use secure libraries.

4.2.2 Static analysis

Static analysis techniques analyze program code including source code, bytecode, or binary code to learn how the control or data would flow at runtime without running the code. Due to the complexity and technical limitations, some static analysis techniques cannot detect the existence of input validation routines and result false positives. Pixy [15] performs tainted data flow analysis using flow-sensitive, interprocedural, context-sensitive data flow analysis and checks if user input is used at a target statement without any input validation. WebStaticApproximation[n] [29] uses a static string analysis

technique [7] to approximate possible string output for variables in a web application and checks if the approximated string output is disjoint with unsafe strings defined in a specification file. If the approximate string output is disjoint with the unsafe strings, WebStaticApproximation^n reports that the application is not vulnerable.

4.2.3 Dynamic analysis

Dynamic analysis techniques analyze the information obtained during program execution to detect vulnerabilities. Dynamic analysis is performed at testing time during development or runtime after software release. SQLGuard [6] and SQLCheck [31] add a special sequence of code before and after user input to distinguish user input portions in a dynamically-constructed SQL query at runtime and check grammar-based violation. For example, if the syntactic structures of an SQL query with and without user input differ, the application is vulnerable. Another example of dynamic analysis is anomaly detection that detects deviation from normal behavior at runtime [10, 12, 32]. The disadvantage of dynamic analysis is that only the vulnerabilities in the execution paths are detected and therefore cannot detect vulnerabilities in parts of the code that were not executed.

Dynamic analysis can be performed after static analysis to use the result of static analysis to improve the efficiency of dynamic analysis. SQLStoredProcedure^n [33] statically performs control flow analysis to reduce the number of SQL statements to verify at runtime. AMNESIA [8] statically builds an SQL query automaton that represents all the possible SQL queries at a target statement using string analysis [7]. AMNESIA compares the statically-generated SQL query automaton with a dynamically-constructed SQL query and detects the mismatch in the structure of the SQL query at runtime. PQLMatcher^n [27] detects the flow of objects that matches the patterns described by a developer in PQL. Static analysis is performed to limit code instrumentation only to the relevant code and improve the performance in PQLMatcher^n, and then dynamic analysis is performed to find the flow of objects

that matches a specified pattern at runtime.

4.2.4 Black-box testing

Black-box testing detects vulnerabilities by testing applications based on requirements specification without knowing the internal structure of source code. Automated scanning techniques such as SecuBat [16] and WAVES [12] are in this category. Automated scanning techniques gather web pages of a given web site using an agent called a web crawler, inject test input and observe the results of execution. SecuBat [16] injects a single quotation mark as test input to detect an SQL injection vulnerability. If the response from a web request includes an `SQLException` error, the error indicates that the user input reached an SQL query statement and the single quotation mark was interpreted as a part of SQL command not as a normal user input. Therefore the program is vulnerable to SQL injection attacks. WAVES [12] also detects error messages that indicate the user input reaches an SQL query statement without being validated.

4.2.5 White-box testing

White-box testing detects vulnerabilities by testing applications with test cases generated based on internal structure of source code. XSSTestGen^n [24] is a white-box testing technique based on static analysis. XSSTestGen^n statically analyzes control flow of web applications and identifies vulnerable, potentially vulnerable and non-vulnerable programs and generates test cases for vulnerable and potentially vulnerable programs.

4.2.6 Summary of literature

Table 4 shows the literature for detection methods.

Table 4. Literature according to analysis methods

Analysis methods	XSS	SQL injection	Both
Secure programming		[28] (manual)	[23] (automated)

Static analysis	[29]		[15]
Dynamic analysis	[10], [12], [14], [18]	[2], [6], [8], [9] [33]	[5, 17], [13], [22, 32], [27], [30], [31]
Black-box testing	[16]	[12], [16]	
White-box testing	[24]		

4.3 Detection time

When to apply a vulnerability detection technique should be determined before an organization chooses an appropriate detection technique. Vulnerabilities can be detected at coding time, testing time, or operation time in the field. This section explains each detection time.

4.3.1 Coding time

Coding time techniques allow early detection of vulnerabilities and therefore reduce the cost of fixing caused by later detection [4]. Static analysis techniques can detect vulnerabilities at coding time without executing code [15, 29].

4.3.2 Testing time

SecuBat [16] and WAVES [12] are black-box testing techniques that test applications without knowing the internal structure of code. XSSTestGen^n [24] is a white-box testing technique that uses the internal structure of code obtained by static analysis to identify the flow of user input and to find potentially vulnerable server programs and input variables that cause the vulnerabilities. XSSTestGen^n generates test cases to generate test input to verify the actual vulnerability identified by static analysis.

4.3.3 Operation time

Operation time techniques detect vulnerabilities at runtime in the field after software is released. Many operation time detection techniques prevent attacks by stopping the execution or changing the malicious input to a non-malicious input after malicious input is detected. However, when operation

time detection techniques have false positives, stopping the execution can cause significant inconvenience to users.

4.3.4 Summary of literature

Table 5 shows the papers in each category.

Table 5. Literature according to lifecycle time of usage

Detection time	XSS	SQL injection	Both
Coding time	[29]		[15], [23]
Testing time	[16], [24]	[12], [16]	
Operation time	[10], [12], [14], [18]	[2], [6], [8], [9], [28], [33]	[5, 17], [13], [22, 32], [27], [30], [31]

4.4 Detection locations

Application-level vulnerabilities are detected at various locations in a system. We classified detection locations into six categories: server-side proxy, server-side application, client-side application, browser, and client-side proxy.

4.4.1 Server-side proxy

Server-side proxy techniques use an additional server or a gateway between a client and a web server. A server intercepts users' requests and analyzes if the input is malicious. If a request includes malicious input, the server rejects the request or changes the malicious input into a non-malicious input. For example, AUSELSQI [2] intercepts a HTTP request at an intermediate server before the request is forwarded to a web server, and checks if the input includes suspicious characters. If the request includes suspicious characters, the intermediate server rejects the request.

4.4.2 Server-side application

Server-side application techniques statically or dynamically analyze server-side applications written

in script languages such as PHP and JSP or programming languages such as Java or C using the methods described in Section 4 to detect vulnerabilities. For example, Pixy [15] statically analyzes PHP scripts to detect XSS vulnerabilities via tainted data flow analysis. AMNESIA [8] statically analyzes Java byte code and constructs automata for SQL queries from the byte code and compares the statically-constructed automata with dynamically-constructed SQL queries at runtime to detect SQL injection attacks.

4.4.3 Client-side application

Client-side application techniques analyze client side scripts and HTML pages to detect vulnerabilities. For example, SecuBat [16] analyzes HTML pages, including client side scripts, and inserts test data to the input fields of forms found from the HTML pages.

4.4.4 Browser

JavaScriptMozilla[n] [10] detects XSS at the browser-level by modifying an existing browser to observe the behavior of scripts.

4.4.5 Client-side proxy

DetectCollectXSS[n] [14] copies the input included in the user request at a client-side web proxy before a request is sent to a web server. If the input includes an executable script and the response includes the same script copied at the proxy, then the vulnerability is detected. Noxes [18] uses a client side web proxy to filter a web request that violates the security policy defined by users. The security policy consists of web site domains that a user permits the browser to access.

4.4.6 Summary of literature

Table 6 shows the papers categorized in each detection location.

Table 6. Literature according to detection location

Detection location	XSS	SQL injection	Both
Server-side proxy	[12]	[2], [33]	[5, 17], [22, 32], [30], [31]
Server-side application	[24], [29]	[6], [8], [9], [28]	[13], [15], [23], [27]
Client-side application	[16]	[12], [16]	
Browser	[10]		
Client-side proxy	[14], [18]		

4.5 Reaction after detection

We identified five categories on the actions taken by techniques after XSIVs are detected. Some techniques just report the fact that vulnerabilities are found with the line number of vulnerable location in the source code. Static analysis and testing techniques are in this category.

Some techniques reject user requests and block the malicious input, while other techniques escape the input to change the malicious input to non-malicious input. However, escaping malicious input can still allow attacks. For example, when the escaped user input is stored to a database and reused later, it can be used for an SQL injection attack [3]. Some techniques use user-defined actions. For example, developers can write an input-escaping method that is executed when PQLMatcher^{n} [27] detects a user input flow that matches a predefined object flow pattern at runtime. With WebSecurityAbstraction^{n} [30], users specify validation constraints and transformation rules. The transformation rules define what to do when a certain pattern is found from user input. For example, programmers can define a rule to escape user input or change the input into HTML-encoding. Noxes [18] asks users to determine if the user will allow the browser to access to a link or not when the link accesses an external site. In this case, users should be able to distinguish whether the link access will result in an XSS attack or not. Table 7 shows the papers in each category of reaction.

Table 7. Literature according to reaction after detection

Reaction after detection	XSS	SQL injection	Both
Reporting	[12], [16], [24], [29]	[12], [16]	[15], [22, 32]
Rejecting	[10]	[2], [8], [9], [33]	[5, 17], [31]

Escaping	[14]	[28]	[23]
User defined action		[6]	[13], [27], [30]
Asking user interactively	[18]		

5. EVALUATION CRITERIA

Once the types of vulnerabilities and the lifecycle time to use the techniques have been determined by an organization, important factors to guide the selection of techniques are accuracy, execution time, and ease-of-use. The evaluation criteria dimension provides a way of classifying the techniques in terms of the presentation of accuracy and the ease-of-use. We did not include execution time in this survey due to the lack of information in the literature.

5.1 Accuracy

Accuracy of detection techniques can be measured by false positive rate and false negative rate. A false positive occurs when a technique generates an alarm for vulnerability detection when there is no actual vulnerability. A false negative occurs when a technique does not detect the type of vulnerabilities that the technique was supposed to detect. False positive rate is the percentage of false positives among total alerts. False negative rate is the percentage of false negatives among total vulnerabilities. False negative rate is difficult to be measured because identification of all the vulnerabilities is impossible due to ever-evolving attack patterns and because the change in the environment of software operation can create new vulnerabilities.

However, some techniques measured false negatives by their own definition of false negative. For example, AMNESIA [8] created a set of illegal input that was ensured to be malicious by an expert. Then, the authors checked if their tools detected all the illegal input identified by the expert. Using their definition of false negative, AMNESIA had no false negatives. SQLCheck [31] provided a theoretical proof of no false negatives and no false positives in the approach. SQLCheck also

performed the evaluation of accuracy of the implemented tool using the same approach as AMNESIA. Organizations should keep in mind that there can be differences in the technique and the implementation of technique due to the complexity in implementation.

WAVES [12] calculates the probability of false negatives rather than measuring the actual false negative rate from evaluation. To calculate the probability of false negative rate, WAVES uses the probability that WAVES can successfully fill a form in a web application with test input and the probability that WAVES correctly distinguish actual vulnerabilities from test execution results.

Knowing the definition of accuracy in each detection technique is important to compare and select appropriate detection techniques. However, many techniques we reviewed did not mention accuracy or clearly provide a way to measure it. We classified the presentation of false positives and false negatives into three groups depending on how each paper described accuracy. *Concrete presentation* means that the paper on a technique described the accuracy with concrete values such as the number of false positives or false negatives from evaluation results. *Explicit presentation* means that the paper on a technique explicitly discussed false positives or false negatives without any evaluation results. *Implicit presentation* means that the paper on a technique did not mention false positives and false negatives explicitly, but we can infer the factors that could cause false positives or false negatives from the limitations described in the paper or from the technique used. Table 8 shows papers in each category of presentation of false positives.

Table 8. Literature according to presentation of false positives

Presentation of false positives	**XSS**	**SQL injection**	**Both**
Concrete		[8] (0%), [9] (0%)	[13] (26.9%), [15] (50%), [22, 32] (< 0.2%), [31] (0%)
Explicit		[2], [33]	[27]
Implicit	[10], [12], [14], [16], [18], [24], [29]	[6], [12], [16], [28]	[5, 17], [23], [30]

Grammar-based approaches typically assume that applications do not allow SQL keywords as normal user input and therefore user input containing SQL keywords are considered malicious. However, if an application allows SQL keywords as normal user input, these tools can generate false positives. Input signature detection can generate false positives when input signatures are too restrictive. Output signature detection can generate false positives when output from execution includes predefined keywords such as a certain error message when there is actually no such error. Additionally, incorrect specification of trusted input source [9] or input string [29], or lack of necessary rules [30] can generate false positives. Tainted data flow detection techniques report false positives when the distrusted input sources are not listed completely [15]. Anomaly detection techniques have false positives when the training data was not enough to comprehensively represent normal behavior. False positives also can occur when the analysis of source code is incomplete due to technical limitations of static analysis. For example, Pixy [15] does not support object-oriented features and assumes that data from object member variables and methods are malicious.

Table 9 shows papers in each category of presentation of false negatives. False negatives can happen in a black-box testing approach when a tool fails to collect exhaustive web pages [12, 16]. An incomplete list of malicious patterns [2], an incomplete specification of unsafe string or distrusted input sources [27, 29, 31], and incorrect policy description [30] also can lead to false negatives.

Table 9. Literature according to presentation of false negatives

Presentation of false negatives	XSS	SQL injection	Both
Concrete		[8] (0%), [12] (2.46%, for SQL injection)	[31] (0%)
Explicit		[9]	[27]
Implicit	[10], [12], [14], [16], [18], [24], [29]	[2], [6], [16], [28], [33]	[5, 17], [13] (for XSS), [15], [22, 32], [23], [30]

Many of the reviewed papers did not provide concrete or explicit presentation of accuracy. A

standard of accuracy measurement and precise presentation of accuracy can be a future work for the research community.

5.2 Ease-of-use

XSIV detection techniques should be easily adopted by organizations and developers with as little effort as possible. Ease-of-use can be measured by required expertise or additional operational complexity in using the techniques. We identified nine categories for additional efforts. Some detection techniques require programming efforts by using special routines. For example, SQL DOM [28] requires developers to use predefined class methods generated for each schema to construct SQL queries. Some techniques require additional specification to specify trusted input sources [9], distrusted input sources [23], or user-defined transformation rules [30]. Some detection techniques require multiple types of additional efforts. Some techniques that require a list of trusted/distrusted input sources or malicious characters/keywords would not need any additional efforts to describe the list because the list is already included in a default configuration of the tools or hard coded. Even in that case, the list should be updated as more input sources are added and more attack patterns are discovered. Therefore, we assigned those techniques into the additional specification category.

Some techniques require a server or client side proxy. Also, some techniques require special operational environment that must be provided by software vendors rather than by an organization that is developing an application. SQLStoredProcedure^{n} [33] requires new database front-end to process stored procedures. JavaScriptMozilla^{n} [10] requires a special browser to intercept and analyze execution of JavaScript. Noxes [18] requires users to define initial filtering rules and add new rules interactively. WebSSARI [13] requires programmers to use an extended script language to express type qualifier. Anomaly detection techniques require a model to be created from execution

with non-malicious data during training period. Testing techniques require test input. Table 10 shows papers in each category of additional efforts.

Table 10. Literature according to additional efforts

Additional efforts	XSS	SQL injection	Both
Application programming		[6], [28]	[15], [31]
Additional specification	[10], [29]	[9], [12], [16]	[13], [15], [23], [27], [30]
Server-side configuration		[2]	[5, 17], [30], [31]
Client-side configuration	[14], [18]		
Special operational environment	[10] (special browser)	[33] (special database front-end)	
User interaction	[18]		
Language extension			[13]
Data training	[12]		[22, 32]
Test input generation	[16], [24]	[12], [16]	
None		[8]	

6. Conclusion and Future Work

We created and presented a taxonomy of vulnerability detection techniques for XSS and SQL injection to provide a basis for comparison and selection of techniques. We reviewed 23 techniques that deal with XSIVs from 21 papers. Our survey shows that the detection techniques are distributed in a wide range of detection criteria, analysis methods, and detection locations. Our taxonomy provides categories of techniques that organizations must consider when analyzing the effectiveness and efficiency of vulnerability detection techniques and also reveals that organizations can have false sense of security without clear understanding of techniques. Different detection criteria and analysis methods result in different accuracy and coverage of vulnerability detection and our taxonomy helps to understand those differences. Techniques also use different precision in presenting the accuracy of their techniques. Measures and presentation of accuracy should be standardized as a cooperative effort of researchers. Measuring accuracy of different techniques using the same benchmark with known vulnerabilities can help the fair comparison of vulnerability

detection. SecuriBench[6] is one of these approaches.

Security can be improved by an efficient combination of techniques. Further research on how to quantify the effectiveness and efficiency of the techniques to maximize the utility of combinations of techniques is required. Organizations and projects have their own risks and requirements on security. Therefore, the quantification must be able to rank the techniques to be used for an organization reflecting the risks and requirements on security.

[6] http://suif.stanford.edu/~livshits/securibench/

APPENDIX A. TAXONOMY CATEGORIZATIONS OF TECHNIQUES

This appendix provides a table of detection techniques with categories according to our taxonomy described in this paper. In Table 11, FP stands for false positive and FN stands for false negative.

Table 11. Summary of TeXSIVs

		Detection Method					Evaluation Criteria	
Technique	**Vulnerability Type**	**Detection criteria**	**Analysis method**	**Detection time**	**Detection locations**	**Reaction after detection**	**Accuracy - Presentation of FP and FN**	**Ease-of-use – Additional efforts**
AUSELSQI [2]	SQL	Input signature	Dynamic analysis	Operation time	Server-side proxy	Reject	FP: Explicit FN: Implicit	Server-side configuration
SQLRand[n] [5, 17]	Both	Grammar-based	Dynamic analysis	Operation time	Server-side proxy	Reject	FP: Implicit FN: Implicit	Server-side configuration
SQLGuard [6]	SQL	Grammar-based	Dynamic analysis	Operation time	Server-side application	User defined action	FP: Implicit FN: Implicit	Application programming
AMNESIA [8]	SQL	Grammar-based	Dynamic analysis	Operation time	Server-side application	Reject	FP: Concrete FN: Concrete	None
WASP [9]	SQL	Taint data flow + Grammar-based	Dynamic analysis	Operation time	Server-side application	Reject	FP: Concrete FN: Explicit	Additional specification
JavaScriptMozilla[n] [10]	XSS	Anomaly	Dynamic analysis	Operation time	Browser	Reject	FP: Implicit FN: Implicit	Special operational environment + Additional specification
WAVES [12]	XSS	Anomaly, output signature	Dynamic analysis	Operation time	Server-side proxy	Report	FP: Implicit FN: Implicit	Data training
WAVES [12]	SQL	Output signature	Black-box testing	Testing time	Client-side application	Report	FP: Implicit FN: Concrete	Additional specification + Test input

		Detection Method					Evaluation Criteria	
Technique	**Vulnerability Type**	**Detection criteria**	**Analysis method**	**Detection time**	**Detection locations**	**Reaction after detection**	**Accuracy - Presentation of FP and FN**	**Ease-of-use – Additional efforts**
								generation
WebSSARI [13]	Both	Tainted data flow	Dynamic analysis	Operation time	Server-side application	User defined action	FP: Concrete FN: Implicit	Additional specification + Language extension
DetectCollectXSSn [14]	XSS	Input/ Output signature	Dynamic analysis	Operation time	Client-side proxy	Escape	FP: Implicit FN: Implicit	Client-side configuration
Pixy [15]	Both	Tainted data flow	Static analysis	Coding time	Server-side application	Report	FP: Concrete FN: Implicit	Additional specification + Application programming
SecuBat [16]	XSS	Expected result violation	Black-box testing	Testing time	Client-side application	Report	FP: Implicit FN: Implicit	Test input generation
SecuBat [16]	SQL	Output signature	Black-box testing	Testing time	Client-side application	Report	FP: Implicit FN: Implicit	Additional specification
Noxes [18]	XSS	Restricted link access	Dynamic analysis	Operation time	Client-side proxy	Ask users	FP: Implicit FN: Implicit	Client-side configuration + User interaction
AnomalyDetectionn [22, 32]	Both	Anomaly	Dynamic analysis	Operation time	Server-size proxy	Report	FP: Concrete FN: Implicit	Data training
AntiMaliciousInjectionn [23]	Both	Input signature	Secure programming	Coding time	Server-side application	Escape	FP: Implicit FN: Implicit	Additional specification
XSSTestGenn [24]	XSS	Tainted data flow + Expected result violation	White-box testing	Testing time	Server-side application	Report	FP: Implicit FN: Implicit	Test input generation
PQLMatchern [27]	Both	Tainted data flow	Dynamic analysis	Operation time	Server-side application	User defined action	FP: Explicit FN: Explicit	Additional specification
SQL DOM [28]	SQL	Input signature	Secure programming	Operation time	Server-side application	Escape	FP: Implicit FN: Implicit	Application programming
WebStaticApproximationn [29]	XSS	Input signature	Static analysis	Coding time	Server-side application	Report	FP: Implicit FN: Implicit	Additional specification
WebSecurityAbstractionn [30]	Both	Input signature	Dynamic analysis	Operation time	Server –side proxy	User defined action	FP: Implicit FN: Implicit	Additional specification + Server-side

		Detection Method					Evaluation Criteria	
Technique	**Vulnerability Type**	**Detection criteria**	**Analysis method**	**Detection time**	**Detection locations**	**Reaction after detection**	**Accuracy - Presentation of FP and FN**	**Ease-of-use – Additional efforts**
								configuration
SQLCheck [31]	Both	Grammar-based	Dynamic analysis	Operation time	Server –side proxy	Reject	FP: Concrete FN: Concrete	Application programming + Server-side configuration
SQLStoredProcedure[n] [33]	SQL	Grammar-based	Dynamic analysis	Operation time	Server –side proxy	Reject	FP: Explicit FN: Implicit	Special operational environment

For easy reference, we used representative names for the techniques when necessary. We used the names from the titles of papers and names of tools described in the papers. However, we associated names we created that can remind the approaches in the cases when no specific names were given in the papers. The names we created are notated with superscript [n].

ACKNOWLEDGMENT

This work is supported by the National Science Foundation under CAREER Grant No. 0346903 and Cybertrust Grant No. 1716176. Any opinions expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. We would like to thank the North Carolina State University Software Engineering Realsearch Group for their careful review of and helpful suggestions for the paper.

REFERENCES

[1] *National Vulnerability Database,* http://nvd.nist.gov
[2] A. A. Alfantookh, "An Automated Universal Server Level Solution for SQL Injection Security Flaw," in *Proceedings of the 2004 International Conference on Electrical, Electronic and Computer Engineering (ICEEC '04)*, Cairo, Egypt, 2004, pp. 131 - 135.
[3] *Advanced SQL Injection in SQL Server Applications,* http://www.ngssoftware.com
[4] B. W. Boehm, *Software Engineering Economics*. Englewood Cliffs, NJ, U.S.A.: Prentice-Hall, Inc., 1981.
[5] S. W. Boyd and A. D. Keromytis, "SQLrand: Preventing SQL Injection Attacks," in *Proceedings of the 2nd International Conference on Applied Cryptography and Network Security (ACNS'04)*, Yellow Mountain, China, 2004, pp. 292--302.
[6] 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'05)*, Lisbon, Portugal, 2005, pp. 106 - 113.
[7] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise Analysis of String Expressions," in *Proceedings of the 10th International Static Analysis Symposium (SAS'03)*, San Diego, CA, U.S.A., 2003, pp. 1–18.
[8] W. G. J. Halfond and A. Orso, "AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks," in *Proceedings of 20th ACM International Conference on Automated Software Engineering (ASE'05)*, Long Beach, CA, U.S.A., 2005, pp. 174 - 183.
[9] W. G. J. Halfond, A. Orso, and P. Manolios, "Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering SIGSOFT '06/FSE-14*, Portland, OR, USA, 2006, pp. 175 - 185.
[10] O. Hallaraker and G. Vigna, "Detecting Malicious JavaScript Code in Mozilla," in *Proceedings 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2005)*, Shanghai, China, 2005, pp. 85 - 94.
[11] M. Howard and D. LeBlanc, *Writing Secure Code*. Redmond, WA: Microsoft Press, 2003.
[12] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai, "Web Application security Assessment by Fault Injection and Behavior Monitoring," in *Proceedings of the 12th International Conference on World Wide Web*, Budapest, Hungary, 2003, pp. 148 - 159.
[13] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing Web Application Code by Static Analysis and Runtime Protection," in *Proceedings of the 13th International Conference on World Wide Web*, New York, NY, U.S.A., 2004, pp. 40 - 52.
[14] O. Ismail, M. Etoh, Y. Kadobayashi, and S. Yamaguchi, "A Proposal and Implementation of Automatic Detection/Collection System for Cross-site Scripting Vulnerability," in *Proceedings of the 18th International Conference on Advanced Information Networking and Applications (AINA 2004)*, Fukuoka, Japan, 2004, pp. 145 - 151.
[15] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities," in *2006 IEEE Symposium on Security and Privacy*, Oakland, CA, U.S.A., 2006, pp. 258 - 263.
[16] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic, "SecuBat: A Web Vulnerability Scanner," in *Proceedings of the 15th International World Wide Web Conference*, Edinburgh, UK, 2006, pp. 247 - 257.
[17] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering Code-injection Attacks with Instruction-set Randomization," in *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)*, Washington, DC, U.S.A., 2003, pp. 272 - 280.
[18] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks," in *Proceedings of the 2006 ACM Symposium on Applied Computing* Dijon, France, 2006, pp. 330 - 337.
[19] S. Kost, "Introduction to SQL Injection Attacks for Oracle Developers," Integrity Corporation, 2004.
[20] D. Kristol and L. Montulli, "Internet RFC 2965: Http state management mechanism," October 2000.
[21] I. V. Krsul, *Software Vulnerability Analysis*, Ph.D. Thesis, Purdue University, 1998.
[22] C. Kruegel and G. Vigna, "Anomaly Detection of Web-based Attacks," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, Washington, DC, USA, 2003, pp. 251 - 261.
[23] J.-C. Lin and J.-M. Chen, "An Automatic Revised Tool for Anti-Malicious Injection," in *Proceedings of the Sixth IEEE International Conference on Computer and Information Technology (CIT '06)* Seoul, Korea, 2006, pp. 164-169.
[24] G. A. D. Lucca, A. R. Fasolino, M. Mastroianni, and P. Tramontana, "Identifying Cross Site Scripting Vulnerabilities in Web Applications," in *Proceedings Sixth IEEE International Workshop on Web Site Evolution (WSE 2004)*, Chicago, IL, U.S.A., 2004, pp. 71-80.
[25] O. Maor and A. Shulman, "Blindfolded SQL Injection," Imperva Inc., 2003.
[26] O. Maor and A. Shulman, "SQL Injection Signatures Evasion," Imperva Inc., 2004.

[27] M. Martin, B. Livshits, and M. S. Lam, "Finding Application Errors and Security Flaws using PQL: a Program Query Language," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, San Diego, CA, U.S.A., 2005, pp. 365-383.
[28] R. A. McClure and I. H. Kruger, "SQL DOM: Compile Time Checking of Dynamic SQL Statements," in *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, St. Louis, MO, U.S.A., 2005, pp. 88–96.
[29] Y. Minamide, "Static Approximation of Dynamically Generated Web Pages," in *Proceedings of the 14th International Conference on World Wide Web*, Chiba, Japan, 2005, pp. 432–441.
[30] D. Scott and R. Sharp, "Abstracting Application-Level Web Security," in *Proceedings of the 11th International Conference on World Wide Web*, Honolulu, Hawaii, 2002, pp. 396 - 407.
[31] Z. Su and G. Wassermann, "The Essence of Command Injection Attacks in Web Applications," in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06)*, Charleston, SC, U.S.A., 2006, pp. 372 - 382.
[32] F. Valeur, D. Mutz, and G. Vigna, "A Learning-Based Approach to the Detection of SQL Attacks," in *Proceedings of the Conference on Detection of Intrusions and Malware Vulnerability Assessment (DIMVA)*, Vienna, Austria, 2005, pp. 123-140.
[33] K. Wei, M. Muthuprasanna, and S. Kothari, "Preventing SQL Injection Attacks in Stored Procedures," in *Proceedings of the 2006 Australian Software Engineering Conference (ASWEC'06)*, Sydney, Australia, 2006, pp. 191-198.

Yonghee Shin is a Ph.D. student in Computer Science at North Carolina State University. She received her MS in Computer Science from Texas A&M University, and her BS in Computer Science from Sookmyung Women's University in Seoul, Korea. She worked for eight years in industries before she returns to academia for her MS degree. Her research interest is in software reliability, software security testing, and software security metrics. Her e-mail address is yonghee.shin@ncsu.edu.
Laurie Williams is an Associate Professor in the Computer Science Department of the College of Engineering at North Carolina State University. She leads the Software Engineering Realsearch group and is also the Director of the North Carolina State University Laboratory for Collaborative System Development and the Center for Open Software Engineering. Laurie is also a certified instructor of the software reliability engineering course, More Reliable Software Faster and Cheaper. Laurie received her Ph.D. in Computer Science from the University of Utah, her MBA from Duke University, and her BS in Industrial Engineering from Lehigh University. She worked for IBM for nine years in Raleigh, NC before returning to academia. Her email address is willams@csc.ncsu.edu.