A Taxonomy of SQL Injection Defense Techniques

Anup Shakya, Dhiraj Aryal
This thesis is submitted to the School of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Computer Science. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author(s): Anup Shakya
Address: Älgbacken 8, 372 34 Ronneby
E-mail: anup007np@gmail.com

Author(s): Dhiraj Aryal
Address: Lindblomsvägen 96, 372 33 Ronneby
E-mail: dhiraj_asp@hotmail.com

University advisor(s): Dr. Stefan Axelsson
School of Computing
Blekinge Institute of Technology

School of Computing
Blekinge Institute of Technology
SE – 371 79 Karlskrona
Sweden

Internet : www.bth.se/com
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57
Abstract

Context: SQL injection attack (SQLIA) poses a serious defense threat to web applications by allowing attackers to gain unhindered access to the underlying databases containing potentially sensitive information. A lot of methods and techniques have been proposed by different researchers and practitioners to mitigate SQL injection problem. However, deploying those methods and techniques without a clear understanding can induce a false sense of security. Classification of such techniques would provide a great assistance to get rid of such false sense of security.

Objectives: This paper is focused on classification of such techniques by building taxonomy of SQL injection defense techniques.

Methods: Systematic literature review (SLR) is conducted using five reputed and familiar e-databases; IEEE, ACM, Engineering Village (Inspec/Compendex), ISI web of science and Scopus.

Results: 61 defense techniques are found and based on these techniques, a taxonomy of SQL injection defense techniques is built. Our taxonomy consists of various dimensions which can be grouped under two higher order terms; detection method and evaluation criteria.

Conclusion: The taxonomy provides a basis for comparison among different defense techniques. Organization(s) can use our taxonomy to choose suitable owns depending on their available resources and environments. Moreover, this classification can lead towards a number of future research directions in the field of SQL injection.

Keywords: SQL injection, Defense technique, Taxonomy, Security, Web application
Acknowledgements

This research would not have been possible without the sincere help and contributions of the many individuals involved directly or indirectly at different levels of our work. We would like to use this opportunity for expressing our sincere gratitude to them. First, we would like to thank our supervisor Dr. Stefan Axelsson for providing us his invaluable guidance, enduring support and advises throughout the research. We also, would like to ponder our thankfulness to our librarian Miss Sofia Swartz for her guidance on finding related papers. Finally yet importantly, we are deeply indebted to our parents back in Nepal and our BTH colleagues for their unconditional support and encouragement due to which we are able to wound up all the challenges we faced.
# Table of Contents

Abstract ....................................... iii
Acknowledgements ........................................ iv
Table of Contents ....................................... v
List of Tables ........................................ xii
List of Figures ......................................... xiii

## Chapter

1 Introduction ........................................ 1
2 Background on SQLIA ............................... 4
3 Previous Work ....................................... 10
4 Research Methodology .............................. 12
   4.1 Literature review ................................ 12
   4.2 Systematic review ................................ 13
5 Planning The Review ................................ 14
   5.1 The need for a review ............................. 14
   5.2 Specifying the research aim ....................... 14
   5.3 Developing a review protocol ....................... 14
      5.3.1 Search Strategy .......................... 15
6 Conducting The Review ............................ 18
   6.1 Selection of primary studies ....................... 18
      6.1.1 Inclusion/Exclusion criteria ................ 18
      6.1.2 Quality assessment criteria ................ 19
7 Taxonomy of SQL Injection Defense Techniques ............. 23
   7.1 Overview ....................................... 23
8 Detection Method ................................... 27
8.1 Classification by nature of defense ................................. 27
  8.1.1 Prevention .................................................. 28
  8.1.2 Detection .................................................... 28
  8.1.3 Deflection ................................................... 28

8.2 Classification by detection principle ................................. 28
  8.2.1 Grammar-based violation detection ............................ 30
  8.2.2 Signature based detection .................................... 30
  8.2.3 Tainted data flow detection .................................. 31
  8.2.4 Anomaly detection .......................................... 32

8.3 Classification by analysis method .................................... 35
  8.3.1 Secure programming .......................................... 35
  8.3.2 Static analysis ............................................... 35
  8.3.3 Dynamic analysis ............................................ 37
  8.3.4 Hybrid analysis ............................................. 37
  8.3.5 Black box testing ............................................ 37
  8.3.6 White box testing .......................................... 38

8.4 Classification by detection time ..................................... 38
  8.4.1 Coding time .................................................. 38
  8.4.2 Testing time .................................................. 38
  8.4.3 Operation time .............................................. 38

8.5 Classification by detection location .................................. 39
  8.5.1 Client-side application ....................................... 39
  8.5.2 Server-side application ...................................... 39
  8.5.3 Client-side proxy ........................................... 39
  8.5.4 Server-side proxy .......................................... 39

8.6 Classification by response ............................................ 40
  8.6.1 Report ....................................................... 40
  8.6.2 Reject ....................................................... 40

vi
C.2 Web Application Security Assessment by Fault Injection and Behavior Monitoring ................................................................. 92
C.3 An Automated Universal Server Level Solution for SQL Injection Security Flaw ................................................................. 92
C.4 SQLRand: Preventing SQL Injection Attacks ............................................. 93
C.5 JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications ........ 93
C.6 Securing Web Application Code by Static analysis and Runtime Protection ................................................................. 94
C.8 Using Parse Tree Validation to Prevent SQL Injection Attacks .......... 95
C.9 SQL Injection Protection by Variable Normalization of SQL Statement 95
C.10 Dynamic Taint Propagation for Java .................................................. 96
C.11 AMNESIA: Analysis and Monitoring for Neutralizing SQL Injection Attacks ................................................................. 97
C.12 Finding Security Vulnerabilities in Java Applications with Static Analysis ................................................................. 98
C.14 SQL DOM: Compile Time Checking of Dynamic SQL Statements ... 99
C.15 A Learning-Based Approach to the Detection of SQL Attacks ....... 100
C.16 Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities ................................................................. 101
C.17 SecuBat: A Web Vulnerability Scanner ............................................. 101
C.18 Automatic Revised Tool for Anti-Malicious Injection ......................... 102
C.19 Eliminating SQL Injection Attacks - A Transparent Defense Mechanism ................................................................. 103
C.20 Defending Against Injection Attacks through Context-Sensitive String Evaluation ................................................................. 103
C.21 The Essence of Command Injection Attacks in Web Applications  . 104
C.22 Preventing SQL Injection Attacks in Stored Procedures  . 105
C.23 Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications  . 105
C.24 Detecting Malicious SQL  . 105
C.25 Using Aspect Programming to Secure Web Applications  . 106
C.26 SMask: Preventing Injection Attacks in Web Applications by Approximating Automatic Data/Code Separation  . 106
C.27 Sania: Syntactic and Semantic Analysis for Automated Testing against SQL Injection  . 107
C.28 Automated Protection of PHP Applications against SQL-injection Attacks  . 108
C.29 Using Automated Fix Generation to Secure SQL Statements  . 108
C.31 Evaluation of Anomaly Based Character Distribution Models in the Detection of SQL Injection Attacks  . 109
C.32 Automated Fix Generator for SQL Injection Attacks  . 110
C.33 SAFELI: SQL Injection Scanner Using Symbolic Execution  . 110
C.34 WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation  . 111
C.35 SQL-IDS: A Specification-based Approach for SQL-Injection Detection  . 112
C.36 An Automatic Mechanism for Sanitizing Malicious Injection  . 112
C.37 SDriver: Location-specific Signatures Prevent SQL Injection Attacks  . 113
C.38 MUSIC: Mutation-based SQL Injection Vulnerability Checking  . 113
C.39 Design Considerations for a Honeypot for SQL Injection Attacks  . 114
C.40 Combinatorial Approach for Preventing SQL Injection Attacks  . 114
C.41 A Weight-Based Symptom Correlation Approach to SQL Injection Attacks .................................. 115
C.42 An Approach for SQL Injection Vulnerability Detection ........................................... 116
C.43 Automatic Creation of SQL Injection and Cross-Site Scripting Attacks .......................... 116
C.44 SQLProb: A Proxy-based Architecture towards Preventing SQL Injection Attacks ............. 117
C.45 Shielding against SQL Injection Attacks using ADMIRE Model .................................. 118
C.46 A Hybrid Analysis Framework for Detecting Web Application Vulnerabilities ......................... 118
C.47 Multi-Layered Defense against Web Application Attacks ........................................... 119
C.48 SBSQLID: Securing Web Applications with Service Based SQL Injection Detection ............. 120
C.49 Intrusion Detection in Web Applications: Evolutionary Approach ................................ 120
C.50 On Automated Prepared Statement Generation to Remove SQL Injection Vulnerabilities ... 121
C.51 A Novel Injection Aware Approach for The Testing of Database Applications ......................... 121
C.52 CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks ........................................... 122
C.53 TAPS: Automatically Preparing Safe SQL Queries ........................................... 122
C.54 Preventing Injection Attacks with Syntax Embeddings - A Host and Guest Language Independent Approach ........................................... 123
C.55 A Heuristic-based Approach for Detecting SQL-injection Vulnerabilities in Web Applications ........................................... 124
C.56 An Approach to Detection of SQL Injection Attack Based on Dynamic Query Matching .... 124
C.57 Access Control Mechanism for Web Databases by using Parameterized Cursor ......................... 125
C.58 Use of Query Tokenization to Detect and Prevent SQL Injection Attacks

C.59 Artificial Neural Network based Web Application Firewall for SQL Injection

C.60 D-WAV: A Web Application Vulnerabilities Detection Tool using Characteristics of Web Forms

C.61 X-LOG Authentication Technique to Prevent SQL Injection Attacks
List of Tables

5.1 General search strings ...................................... 16
6.1 Inclusion/Exclusion criteria .................................. 21
6.2 Quality assessment criteria .................................. 22
6.3 Summary of total number of selected papers .............. 22
8.1 Classification by detection principles ...................... 29
8.2 Classification by analysis methods ......................... 36
A.1 Summary of classification of defense techniques according to detection method .............................................. 74
B.1 Summary of classification of defense techniques according to evaluation criteria .............................................. 85
List of Figures

1.1 OWASP SQL injection ranking .................................. 2
2.1 How web application works ................................... 5
2.2 SQLIA vulnerable PHP code snippet for login form ............ 6
2.3 Login form with benign user inputs ............................ 7
2.4 Login form with malicious user inputs .......................... 8
7.1 Detection method .............................................. 25
7.2 Evaluation criteria ............................................ 26
8.1 FSM for a SQL query ........................................... 30
10.1 Number of defense techniques over the years ................. 46
10.2 Usage of detection principles ................................. 47
10.3 Usage of detection principles over the years .................. 48
10.4 Usage of analysis methods .................................... 49
10.5 Usage of analysis methods over the year .................... 50
10.6 Level of details in terms of false positive, false negative and performance overhead ........................................... 52
Chapter 1

Introduction

SQL injection attack (SQLIA) poses serious threat to the internet since long time. Many defense techniques have been proposed to tackle it. However, it is still extremely common, even taking the number one spot in the Open Web Application Security Project (OWASP) top ten list published in 2010 [60]. Every three year, OWASP releases top ten lists of most dangerous security flaws in web applications. Figure 1.1 shows the ranking of SQL injection in OWASP top ten lists over the years.

![Figure 1.1: OWASP SQL injection ranking](image)

The existing defense techniques deploy diverse methods to tackle the problem, and it is difficult to understand their advantages and disadvantages, similarities and differences. It is even more challenging to compare among one another and evaluate their cost and efficacy. Without a clear understanding of above mentioned features, the use of any defense technique(s) can generate false security results.

There is a need for up-to-date, systematic survey and taxonomy in the field of SQL injection defense mechanisms. Taxonomy is a classification scheme that helps
us to partition the body of knowledge, and provide us a tool with which to define the relationship and order the complex phenomena into more manageable units. The classification categories of a good taxonomy are unambiguous, exhaustive, and mutually exclusive. It also provides us with the clear ideas about how to explain observed phenomena. By classifying a number of units according to our taxonomy and then observing the possible missing objects i.e. holes, we can exploit the predictive qualities of a taxonomy [5].

In this paper, we propose a taxonomy, together with an up-to-date survey of significant defense techniques. We provide a classification of the defense techniques according to the taxonomy. The taxonomy will provide a foundation for comparing those defense techniques. Organization(s) can use the comparison results in order to choose suitable techniques depending on their available resources and environments. It is important to note that the main focus of our survey is SQL injection defense techniques. It should be noted that authors have provided their techniques with different level of details and hence comparing these techniques based on information from their papers will be a tough task to carry on. Thus, our survey can be used as a foundation to compare different techniques.

We performed the survey with systematic literature review. We have used IEEE, ACM, Scopus, Engineering village (Inspec/Compendex) and ISI web of science digital libraries to collect related papers. We have selected 61 papers related to SQL injection defense techniques ranging from the year 2002 to 2010.

We aim to develop a taxonomy which will emphasize crucial features of SQL defense techniques. Many previous surveys [27, 72, 3] are outdated by today’s standard. In other words, they have missed many potential research approaches due to lack of systematic and taxonomic approach. There is one previous attempt on taxonomy of SQL injection techniques [66]. However, it is devoid of systematic review, hence not exhaustive. It also lacks required depth while classifying detection principles and analysis methods.
Chapter 2

Background on SQLIA

The term "SQL injection" dates back to 1998, while its first public use was in the year 2000 [22]. Since then SQLIA has become one of the most common attack employed over the internet [21]. It occurs when a malicious user modifies the semantic or syntax of a legitimate query by inserting new SQL keywords or operators consequently generating unexpected results not intended by web applications [2]. Injected application does not differentiate between the real and fake queries and displays a result according to what they get in a query, thereby deriving false or unintended results. As a case, Ehud Tenenbaum has been arrested on accusation of stealing 1.5 million dollar from Canadian and at least 10 million dollar from US banks by employing SQLIAs [69]. The threats posed by SQL injection attack may go beyond simple data exploitation. An attacker may also escalate privileges, bypass authentication, execute remote commands to transfer malicious software, or execute a denial-of-service attack [27].

Prior to understand how SQLIA works, it is important to discuss in brief about how web application works. Although web applications come in different shapes and sizes, one thing they have in common, regardless of the programming language in which they were written, is that they are database driven [13]. Database driven web applications are very common in today’s web world. They are simply like black boxes that accept HTTP requests as inputs and create SQL queries as outputs, as illustrated in figure 2.1. The parameter values from HTTP requests are used to generate SQL queries. A back-end database accepts that SQL queries and sends back specific information to the web applications as required by web client in the
As the most common vulnerable page for SQL injection attack is login form, we illustrate a simplified example to show how SQLIA occurs in it. It may be observed that the example symbolizes an exceptionally simple kind of SQLIA, and we portray it only for illustrative purposes. This is the usual query for user login in PHP. Figure 2.2 shows a SQLIA vulnerable PHP code snippet that uses a dynamically generated SQL query to authenticate a user through username and password provided by user in the login form.

```php
// connect to a database
mysql_connect('servername', username, password);

// store user input in the variables collected from the user input login form
$username = $_POST['username'];
$password = $_POST['password'];

// Dynamically build the query from the user input
$query = "SELECT * FROM tbl_users WHERE username = \$username\ AND password = \$password\";

// execute query
$result = mysql_query($query);

if($result)
    return true;
else
    return false;
```

Figure 2.2: SQLIA vulnerable PHP code snippet for login form

Figure 2.3 shows a login form with benign user inputs where password is shown
in normal text rather than a typical "******" format to make example more understandable. Figure 2.4 shows a login form with malicious user inputs.

Figure 2.3: Login form with benign user inputs

The following query is dynamically generated from user inputs as shown in figure 2.3.
SELECT * FROM tbl_users WHERE username = 'david' AND password = 'sweden123';

Figure 2.4: Login form with malicious user inputs

The following query is dynamically generated from malicious user inputs as shown in figure 2.4.
SELECT * FROM tbl_users WHERE username = 'david' OR '1'='1' - 'AND password = 'whatever';
In the above query, everything after double dash "- -" will be ignored by the database because it is a SQL comment operator. The result from the above query will grant malicious user an access regardless of valid username, since result of "1=1" is always true. In this way an attacker can bypass authentication process without a valid username and password. It should be noted that it is not the only way that malicious user can bypass authentication process.
Chapter 3

Previous Work

Though a lot of research has been done in the field of SQL injection, very few studies are focused towards taxonomy of SQL injection attacks and defense techniques. We have found six papers in total which are relevant to the field of our research. Among these papers, only one [66] has worked towards the taxonomy building process of SQLIA defense techniques. In [66] authors present taxonomy of defense techniques that detect SQLIAs as well as cross-site scripting. The survey was not exhaustive and based on only 21 papers published on the ACM and IEEE databases while our research is extensive and systematic. We have reviewed 61 papers which we have collected from five electronic databases (IEEE, ACM, ISI web of science, Engineering village and Scopus). We are more focused towards SQL injection defense techniques.

An extensive work on classification exists [27] in the field of SQL injection and presents informative reading for researcher in the SQL injection defense field. Halfond et al. [27] has contributed major and crucial part in the SQL injection research area by presenting extensive review of different types of SQLIAs. Additionally, it also presents and analyzes existing defense techniques against different SQLIAs with their strengths and weaknesses. However, this paper is not focused towards taxonomy of defense techniques. The two papers [71, 72] are completely based and extends the work of Halfond et al. [27].

In [70] authors present a SQLIA model and classification of SQL injection attacks but have not discussed about defense techniques. In [3] authors present and analyze different SQL injection defense techniques but their survey is limited to only six techniques. Authors have put all their efforts only on the description of each
techniques rather than classification.

To date, major contribution in taxonomy of SQL injection defense techniques has not been exposed; there is a need of such contribution in this research area which we are trying to fill up with our contribution.
Chapter 4

Research Methodology

In this chapter, we discuss the selection of research methods and motivation behind the selection.

4.1 Literature review

Prior to the main review, we performed a literature review to ascertain whether or not a systematic review had been previously conducted in this topic area, to estimate the quantity of research on SQL injection defense techniques, and to gain an indication of the coverage of subjects in the body of work. This included searching for primary studies and existing reviews consistent with the review’s objectives. The literature review helped us to achieve quick and broad knowledge about SQLIAs and SQL injection defense techniques. It encouraged us to find relevant papers in very short span of time and to be familiar with the research area. As, literature review does not guarantee the completeness, bias and systematicity; we conducted literature review as a first step to gain the background knowledge, to identify scope and objectives of the research, and to identify relevant papers for the next steps. The scoping exercise demonstrated that a systematic review of qualitative research of SQL injection defense techniques has, so far, not been conducted yet.
4.2 Systematic review

After getting adequate knowledge about research domain, we performed systematic review to answer our research objectives. Systematic review is suitable for our research as it provides efficient, unbiased and complete results upon which we can provide a basis to draw conclusion and make rational decisions. The systematic review consists of following steps [43]:

- Planning the review
- Conducting the review
- Reporting the review

The detail steps are discussed below.
Chapter 5

Planning The Review

In this chapter, research questions and systematic review protocol are discussed.

5.1 The need for a review

Presently, to the best of our knowledge, such a systematic, taxonomic approach has not been undertaken. Hence, there is a need for up-to-date, thorough and unbiased systematic review in the field of SQL injection defense techniques to efficiently integrate reliable information and provide a basis for rational decision making.

5.2 Specifying the research aim

The main aim of our research is to develop the taxonomy of SQL injection defense techniques and to identify possible directions for future research. In order to achieve this aim, we have formulated following research questions.

RQ1. What are the existing defense techniques against SQLIAs?
RQ2. What are the important attributes for classification of those techniques?

5.3 Developing a review protocol

A review protocol specifies the detail methods that will be undertaken to perform a systematic review which reduces the possibility of researcher bias [16]. In this section, we discuss the components of review protocol.
5.3.1 Search Strategy

Search strategy aims to find both published and unpublished papers. Searches were conducted on core electronic databases: IEEE, ACM digital library, Scopus, Engineering village (Inspec/Compendex) and ISI web of science. We chose these five different electronic databases because of the reputation, familiarity, coverage, exportability and advance search facility [16]. Keywords, titles and abstracts were used as the common search fields. The searches were restricted only to English language papers published from 2002 to 2010.

It should be noted that, it is almost impossible for any search strategy to retrieve 100% of relevant papers due to lack of reporting standards [16]. Therefore, our objective was to perform effective and efficient search strategy so that we won’t miss any potential papers.

As a first step towards construction of string, we reviewed the previous search (literature review). We identified some of the key papers from the literature review and from which we identified the common keywords, terms and phrases to initiate search strings. We revised our search strings to ensure relevance to the review. After several revisions and on subsequent confirmation by supervisor, our search string was ready to perform an actual search. We have refined our search strings by performing number of iterations to achieve desired relevancy and coverage. The search strings were applied on the Title/Abstract/Topic search fields to maximize the number of relevant papers. Table 5.1 shows the general search strings.
<table>
<thead>
<tr>
<th>No.</th>
<th>Strings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;Web application&quot; OR &quot;Web applications&quot; OR &quot;Web Development&quot; OR &quot;Web-apps&quot; OR &quot;Web Security&quot; OR &quot;Web apps&quot;</td>
</tr>
<tr>
<td>2</td>
<td>SQLIAs OR SQLIA OR &quot;SQL Command Injection&quot; OR &quot;Query Poisoning&quot; OR &quot;Malformed SQL&quot; OR &quot;SQL Tampering&quot; OR &quot;Malicious SQL queries&quot; OR &quot;SQL Hacking&quot; OR &quot;SQL Insertion&quot; OR &quot;SQL Insertions&quot; OR &quot;SQL injection&quot; OR &quot;SQL-injection&quot; OR &quot;SQL-injections&quot; OR &quot;SQL injections&quot; OR &quot;SQL intrusion&quot; OR &quot;SQL evasion&quot; OR SIA OR SQLIVs</td>
</tr>
<tr>
<td>3</td>
<td>Prevention OR Vulnerabilities OR Vulnerability OR Detection OR Protection OR Defence OR Defense OR Protecting OR Countermeasure OR Countermeasures OR Block OR Blocking OR Identifying OR Identification</td>
</tr>
<tr>
<td>4</td>
<td>Approach OR Approaches OR Method OR Methods OR Procedure OR Procedures OR Process OR Processes OR Tool OR Tools OR Techniques OR Technique OR Strategy OR Strategies OR Solution OR Solutions</td>
</tr>
</tbody>
</table>

Search Strings: String 1 AND (String 2 AND (String 3 AND String 4))
This is our general search strings which we have used. Some databases like ACM, IEEE have limitations such as number of operators and characters, and advance search facility. Hence, we made necessary changes to the search strings according to the requirements of databases. As an additional search strategy, we scanned the reference list of selected papers so that we won’t miss any potential papers. The search process was applied to search engines like google, msn, and yahoo to find some related papers. All relevant papers are included for our survey.
Chapter 6

Conducting The Review

In this chapter, we discuss about selection of primary studies, inclusion/exclusion criteria and quality assessment criteria.

6.1 Selection of primary studies

The above search strategies yielded total 638 citations from different databases where 294 from ACM, 132 from Engineering village (97 from Compendex and 35 from Inspec), 55 from ISI web of science, 63 from IEEE and 94 from Scopus. We utilized JabRef, open source bibliography reference management software [34], for managing citations, categorizing, storing, sorting of papers, etc. The titles of the citations were read and out of which 484 studies were rejected as these were not concerned with SQL injection. The abstracts of the remaining 154 citations were read thorough again, and a further 44 of the citations were rejected, because these too were not concerned with SQL injection defense techniques and were not apart of qualitative research. In the end, we were left with only 110 citations to study. Many of the abstracts did not provide enough information to be included in the research. Therefore, in order to ascertain whether the research was appropriate, full text articles were obtained for the remaining papers.

6.1.1 Inclusion/Exclusion criteria

We developed a series of inclusion/exclusion criteria to fit in with our review question. The inclusion/exclusion criteria were applied to 110 studies retrieved in the
search strategy. After enforcing inclusion/exclusion criteria, 90 studies were deemed to fit for the review. The inclusion/exclusion criteria for the selection of review are shown in table 6.1.

### 6.1.2 Quality assessment criteria

The quality assessment is an important aspect of any qualitative and systematic review research methodology. There are no commonly approved quality assessment criteria by which to deem the quality of the research. We developed our quality assessment criteria based on modified version of EPPI-Centre criteria, utilized by Elizabeth 2004 [53]. We chose these criteria because these are based on assumption and rationale of qualitative research. The quality assessment criteria are listed in table 6.2. The nine quality assessment criteria were applied independently to each of the 90 studies.

An appraisal system was developed to make the quality assessment criteria transparent. For each of the studies, a label of Yes/No/Partially was awarded, for each of the nine quality assessment criteria. The appraisal system was:

**Yes** – Clearly answer the quality assessment criteria question

**No** – Untrustworthy answer

**Partially** - Imprecise answers which may affect the validity of the conclusion

A final overall assessment was then calculated from each study based on quality assessment criteria. Studies marked as 'Yes, Partially’ were used in the final review; 'No' marked studies were rejected.

Our main aim is to build taxonomy and we believe that the higher the number of diverse defense techniques, the stronger the taxonomy will be. Therefore, we simplified the quality assessment criteria and included 'YES and Partially' marked papers to the review and have not used any grading system to grade the papers. However, we have excluded those papers marked as 'No’, so that we can be sure
of reliability and validity of studies to be synthesized. We found that the papers included for the final review covered a wide range of defense techniques. Table 6.3 shows the summary of the selected papers.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Inclusion criteria</th>
<th>Exclusion criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>Articles published in English.</td>
<td>Articles not in English.</td>
</tr>
<tr>
<td>Authorship</td>
<td>If the authors and content of studies are similar, the most recent version of review or article will be accepted.</td>
<td>Studies by same author similar in content.</td>
</tr>
<tr>
<td>Study Type</td>
<td>Primary research</td>
<td>Book reviews, white paper, policy documents.</td>
</tr>
<tr>
<td></td>
<td>Studies that discuss previous related works.</td>
<td>Studies which do not discuss any previous related works.</td>
</tr>
<tr>
<td>Scope</td>
<td>Studies main focus is on SQL injection defense techniques.</td>
<td>Studies main focus is not on SQL injection.</td>
</tr>
<tr>
<td></td>
<td>Studies concern with SQL injection defense techniques.</td>
<td>Studies do not concern with SQL injection defense technique.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Studies concern with defense techniques other than SQL injection.</td>
</tr>
</tbody>
</table>
Table 6.2: Quality assessment criteria

<table>
<thead>
<tr>
<th>Q.N</th>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Does the study focus on SQL injection defense techniques?</td>
<td>Yes/No/Partially</td>
</tr>
<tr>
<td>2</td>
<td>Does the study clearly state the aims and objectives?</td>
<td>Yes/No/Partially</td>
</tr>
<tr>
<td>3</td>
<td>Does the study adequately describe the context of the research?</td>
<td>Yes/No/Partially.</td>
</tr>
<tr>
<td>4</td>
<td>Are the measures employed in the research sufficiently described?</td>
<td>Yes/No/Partially</td>
</tr>
<tr>
<td>5</td>
<td>Are the measures employed in the research the most pertinent ones for satisfying the research question?</td>
<td>Yes/No/Partially</td>
</tr>
<tr>
<td>6</td>
<td>Does the study provide clear description of the methods used including data collection, data analysis and research framework?</td>
<td>Yes/No/Partially</td>
</tr>
<tr>
<td>7</td>
<td>Does the study discuss the previous related works?</td>
<td>Yes/No/Partially</td>
</tr>
<tr>
<td>8</td>
<td>Does the study discuss any assumptions taken?</td>
<td>Yes/No/Partially</td>
</tr>
<tr>
<td>9</td>
<td>Does the study attempt to validate its findings?</td>
<td>Yes/No/Partially</td>
</tr>
</tbody>
</table>

Table 6.3: Summary of total number of selected papers

<table>
<thead>
<tr>
<th>Steps</th>
<th>Total number of papers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration 1 (After using search string)</td>
<td>638</td>
</tr>
<tr>
<td>Iteration 2 (Selection by title)</td>
<td>154</td>
</tr>
<tr>
<td>Iteration 3 (Selection by abstract)</td>
<td>110</td>
</tr>
<tr>
<td>Iteration 4 (Selection by full text)</td>
<td>90</td>
</tr>
<tr>
<td>Iteration 5 (After quality assessment)</td>
<td>61</td>
</tr>
</tbody>
</table>
Chapter 7

Taxonomy of SQL Injection Defense Techniques

7.1 Overview

SQL injection defense techniques should be able to detect vulnerabilities precisely during development or operation time in the field. The techniques providing higher rate of accuracy and less effort to implement are more likely to be implemented by an organization(s). We created taxonomy for SQL injection defense techniques to help the understanding of accuracy of the techniques and their evaluation criteria including additional infrastructure for fair comparison.

The various dimensions of our taxonomy can be grouped under two higher order terms; detection method and evaluation criteria. The first concerns an aspect of the system while the other concerns an aspect of the evaluation/presentation of the system, not the system itself. Each of these higher order groupings contains several related dimensions. Detection method consists of seven dimensions; nature of defense, detection principle, analysis method, detection time, detection location, response and implementation. Evaluation criteria consists of three dimensions; accuracy (false positive and false negative), performance overhead and additional infrastructure. A particular SQL injection defense technique would thus be given a unique value in each of the dimensions and the classification of the technique as a whole would be described by that point defined by each value for each dimension. Figure 7.1 and 7.2 represent detection method and evaluation criteria respectively.
The dimensions and respective values are explained in detail in chapters 8 and 9. We have seen a rapid and a steady growth in this field of studies. The growth is ever expanding. And because of these facts, the present taxonomy should of course be seen as first attempt. Since there is no consistent terminology, we have difficulty in naming the satisfactory terms for the different classes. Wherever possible we gave new terms for the classes we are trying to define. However, for better understanding and clarity, we at our best, have and tried using the same terms used by the authors in the field. It may create confusion between new terms and old terms, so we tried to give the definition for all the terms we have used. We would like to apologize in advance for any confusion that may arise if the reader already has other definition in the mind for the terms which we have used in this paper.
Figure 7.1: Detection method
Figure 7.2: Evaluation criteria
Chapter 8

Detection Method

Based on the nature of defense, detection principle, and when, where and how SQLIAs are explored and results handled thereafter and how techniques can be implemented, detection methods can be categorized as:

8.1 Classification by nature of defense

Researchers have proposed many different defense techniques to address the problems of SQLIAs. These different techniques can be of different in defense natures. Nature of defense figures out how a technique is going to defend the application from injections. According to the Halme [29], nature of defense for anti-intrusion detection system (IDS) can be classified into six high level categories, namely; prevention, pre-emption, deflection, deterrence, detection and countermeasure. Whereas Halfond [27] has classified nature of defense for SQLIA defense techniques only into two types; detection and prevention. During our survey, we identified three types of defense natures in the area of SQL injection defense techniques. They are prevention, detection and deflection. Preemption and deterrence are natural as they are not as easily translated into technology as the others. Moreover, preemption is considered as too harsh which may be the reason for not implementing it in this research area. Since the prepared statements take care of all the SQL injection deterrence [74], we did not classified deterrence as separate defense nature as classified by Halme [29]. Halme [29] has defined countermeasure as a booby trap while Halfond et al. [27] has used it to refer overall defense techniques against SQLIAs.
8.1.1 Prevention

Prevention technique averts or severely handicaps the possibility of success of SQLIAs by statically identifying vulnerabilities in the code, proposing a different development paradigm for generating SQL queries, or inspecting the application to enforces best defensive coding practices during development [29, 27].

8.1.2 Detection

Detection technique discriminates SQLIA attempts and preparation from benign activity and alerts the system. It detects SQLIAs mostly during the operation time. After detection of attacks, it alerts the authorities so that they can perform certain actions such as rejecting and escaping of attacks [29, 27].

8.1.3 Deflection

Deflection technique leads an attacker to believe that he has succeeded in an injection attempt whereas the reality is that he has been succeeded to compromise false information only. It is designed is such a way that attackers get easily attracted towards it. This technique helps in learning more about different SQLIAs and their attacking customs [29]. Honeypot* [11] is the only one technique that falls under this category.

8.2 Classification by detection principle

Each technique has its own criteria to detect the existence of vulnerabilities in the web application. We have identified four different detection principles: grammar-based violation, signature based, tainted data flow and anomaly detection. Table 8.1 shows the classification by detection principles.
<table>
<thead>
<tr>
<th>Detection Principles</th>
<th>Example Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Grammar based violation</strong></td>
<td>SQLRand [8], Tautology checker* [77] , JDBC Checker [24], AMNESIA [26], SQLGuard [10], SQLCHECK [68], CSSE [59], ESIATDM* [58], StoredProcedure* [78], APPASIA* [54], Syntax Embeddings* [9], Sania [44], AFGSSS* [73], CVSID [80], SecurePHP [17], Prepared Statement [74], SAFELI [23], SQL-IDS [40], WASP [28], SQLInjectionGen [38], SBSQLID [65], SQLProb [48], DUD [15], CANDID [6], D-WAV [79], TAPS [7], Query Parser [45], X-LOG AUTHENTICATION [33], Smask [36], ADMIRE [50], ANNWAF* [57]</td>
</tr>
<tr>
<td><strong>Signature based detection</strong></td>
<td>Input Signature</td>
</tr>
<tr>
<td></td>
<td>Output Signature</td>
</tr>
<tr>
<td><strong>Tainted data flow</strong></td>
<td>Negative Tainting</td>
</tr>
<tr>
<td></td>
<td>Positive Tainting</td>
</tr>
<tr>
<td><strong>Anomaly detection</strong></td>
<td>Learning based</td>
</tr>
<tr>
<td></td>
<td>Program based</td>
</tr>
</tbody>
</table>
8.2.1 Grammar-based violation detection

The grammatical structure of SQL statement is the notion of this detection technique to detect SQL injection vulnerabilities. SQL injection occurs when the attacker provides malicious input that will change the structure of SQL query as intended by the application developer. Grammar-based violation detection technique detects the invalid structure of the SQL statement by comparing the parse tree or finite state machines (FSM) built with user input and without user input. A parse tree can be defined as a data structure for the parsed representation of a SQL statement [10]. If the grammatical structures of parse trees are different, it implies that user input is malicious that will change the indented structure of query and will not allow SQL statement to execute. For example, Figure 8.1 shows Finite State Machine (FSM) representation of a SQL query for the example shown in Figure 2.2.

![Figure 8.1: FSM for a SQL query](image)

SQLIA is detected because FSM built from user input (david’ OR ’1’ = ’1’ –) has different grammatical structure from the one built without user input. A parse tree is built in such a way that any SQL keywords which users included in the user input will create a non unique root. Therefore, the existence of non unique roots in the parse tree of SQL query indicates SQLIA.

8.2.2 Signature based detection

A signature can be as simple as a regular expression describing the known attack pattern. The signature-based detection systems maintain a list of possible attack
signatures, and then compare external input strings against the list of signatures at runtime to detect and block SQL Injection related patterns. The idea of signature based detection techniques is to look for known attack patterns to block. We identified two sub categories of signature based detection techniques.

- **Input signature**

  Input signature detection techniques detect potential malicious characters by checking external input strings against white list or blacklist. White list is a set of safe (possible correct) values where as blacklist is set of unsafe (negative) values. In white list, external input strings are verified against a set of good input values/patterns/conditions and block anything that is not on the white list. In blacklist, external input strings are verified against set of the negative/bad values/patterns/conditions and sanitize the input by user defined action such as rejecting, escaping (adding a backslash) etc. The single quotation mark (‘) is one of an example of blacklist.

- **Output signature**

  Output signature detection techniques detect potential malicious characters from the output of the web application execution before it will be sent to build the SQL query. It is essential to keep in mind that output often contains user input. Secure output handling is important to prevent from SQL injection attack vulnerabilities.

### 8.2.3 Tainted data flow detection

The key idea of tainted data flow detection is to detect whether tainted data will reach sensitive sinks in the application. A tainted data is the input from the user which should always be treated as malicious. Sensitive sinks is any point in the application which could lead to security issues when executed over any un-sanitized
user input. Tainted data flow detection identifies user inputs and also untrustworthy sources and keeps track of all the data that is affected by those input data. Tainted data flow detection can be further divided in two sub categories.

- **Positive tainting**

  Positive tainting approach identifies and mark trusted data instead of untrusted data. It only allows trusted data to form the semantically correct SQL queries such as SQL keywords and operators. In [28] authors proposes positive tainting, which marks trusted data in the source code and make sure that all SQL queries are built only using trusted data.

- **Negative tainting**

  Negative tainting approach identifies and mark un-trusted data instead of trusted data. It basically keeps track of taintedness of data values and checking specifically for malicious contents only in the parts of output that came from un-trustworthy sources [75].

### 8.2.4 Anomaly detection

Anomaly detection techniques triggers alarm when run time behaviour of application diverges from normal system behaviour which was tracked during training period. It is challenging to identify abnormal behaviour of application at run time. The current state of application is periodically compared with the models of the normal system behaviour to detect anomalies. Anomaly detection techniques can only identify attacks which are modelled during training period.

- **Learning based**

  Learning-based anomaly detection approach relies on training dataset to build profiles of the normal, benign behaviour of applications. It commonly uses data mining techniques, clustering techniques to characterize the network traffic and
identify intrusion patterns. Some techniques use statistical analysis to characterize the user behaviour, while other uses artificial neural network (ANN) to train and learn the normal traffic pattern. Some techniques build legitimate libraries while training and detects the attack using that library. In [14] authors described an anomaly based detection approach to detect web application attacks, based on the analysis of the internal state of web application. It learns by analyzing the relationship between internal state of web application and the application’s critical execution points to identify an attack that tries to bring an application in an anomalous state, such as bypassing of the intended work flow of a web application.

- Programmed based

The description of accepted network behaviour is programmed by network administrator or user to detect anomalous events (which fall outside the model of accepted network behaviour). Thus the user defines the rules on what is considered abnormal enough for an application to alert for security violation. Programmed based anomaly detection uses trained specifications of normal behaviour and generate threshold values for different parameters. Such parameters can be the number of network connections, the number of unsuccessful logins etc. Threshold values define whether to raise the alarm or not. For example, alarm if the number of unsuccessful logins is greater than two [5].

**Discussion of detection principles:** Some techniques appear in more than one category; this is not because the classification is ambiguous but because some techniques employ combination of different detection principles. WASP [28], MDWAA* [61], SBSQLID [65] and X-LOG AUTHENTICATION [33] employ several detection principles. The symbol "*" after the name of a techniques represents a name given by us. WASP employ grammar based violation and positive tainted flow detection principles to detect vulnerabilities whilst MDWAA* employ signature based (input
signature) and anomaly detection principles. On the other hand, both SBSQLID and X-LOG AUTHENTICATION employ signature based (input signature) and grammar based violation detection principles.

**Comparison of detection principles** : The accuracy of grammar based violation depends on the algorithm and analysis methods. Input and output signature detection methods can be less accurate than grammar based violation. The dependency of signature based method is on the completion of signatures identified. The main drawback of signature based methods is that only known attacks will be detected, so a signature must be created for every attack. Even a slight variation of old attacks or a new attack may go unnoticed. Signature based methods are also prone to false positives since these are normally based on string matching and regular expressions. However, signature based methods are easy to understand, develop and to implement than grammar based violation [66].

Tainted data flow methods detect only the flow of suspicious data rather than detecting actual potential malicious input. Therefore, tainted data flow can produce false positives and negatives. Tainted data flow methods if combined with other methods such as signature detection methods can be more effective to detect vulnerabilities.

The anomaly detection method relies on the concept of a baseline for normal behaviour constructed during training period with non malicious input. The accuracy of learning based anomaly detection depends on the quality of the training dataset used. A drawback of anomaly detection method lies in the difficultly in defining rules. Another drawback of anomaly detection is that potential malicious activity that falls within normal behaviour patterns is not detected. However, anomaly detection has an advantage over signature based method in that it can discover previously unknown attacks (new attacks) for which a signature does not exist.
8.3 Classification by analysis method

SQL injection detection techniques use several different analysis methods to detect the existence of vulnerabilities in the web application. We identified six different types of analysis methods: secure programming, static analysis, dynamic analysis, hybrid analysis, black box testing and white box testing. Table 8.2 shows classification by analysis methods.

8.3.1 Secure programming

Secure programming is a defensive coding approach to reduce injection vulnerabilities by implementing input validation routines or by using existing standard API or library classes to build the sentence in the source code of application during development. There are lots of secure libraries available provided by vendors. SQL DOM [52] encourages the developers to use a set of classes which is strongly typed to database schema. Instead of using string manipulation to build dynamic SQL queries programmed by developers, uses safe API to generate SQL statements that will take care of security and thus escapes malicious characters to prevent SQL injection attacks. The main drawback of secure programming is that it requires developer training to learn the proper use of secure libraries.

8.3.2 Static analysis

Static analysis techniques analyze applications artifacts such as source code, binary code, byte code, and configuration files in order to get information about an application. Information can be how the data would flow at run time without executing the code. Some techniques rely on complex static analysis in order to identify possible vulnerabilities in the code, for example [77, 26, 49]. Such conservative static analysis can produce high number of false positives.
<table>
<thead>
<tr>
<th>Analysis method</th>
<th>Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure Programming</td>
<td>SQL DOM [52], AntiMaliciousInjection [46], Syntax Embeddings* [9], AFGSSS* [73], SecurePHP [17], Prepared Statement [74]</td>
</tr>
<tr>
<td>Static Analysis</td>
<td>Tautology checker* [77], JDBC Checker [24], Java Static Tainting [49], Pixy [37], CVSID [80], ADMIRE [50]</td>
</tr>
<tr>
<td>Dynamic Analysis</td>
<td>Security Gateway [63], SQLrand [8], AUSELSQI [1], Anomaly detection* [76], UPTI* [75], Java Dynamic Tainting [25], SecuriFly [51], Variable Normalization [62], SQLGuard [10], SQLCHECK [68], CSSE [59], APPASIA* [54], DMSQL* [20], Sania [44], Smask [36], Swaddler [14], AProSec [30], ACMWDPC* [35], AMSMI* [47], CAPSIA* [18], ABCDM* [41], SDriver [55], SQL-IDS [40], WASP [28], WBSCA* [19], Honeypot* [11], Evolutionary Approach* [67], MDWAA* [61], SBSQLID [65], SQLProb [48], IAAT* [4], DUD [15], ANNWAF* [57], CANDID [6], D-WAV [79], TAPS [7], Query Parser [45], X-LOG AUTHENTICATION [33]</td>
</tr>
<tr>
<td>Hybrid Analysis</td>
<td>WebSSARI [32], AMNESIA [26], ESIATDM* [58], StoredProcdure* [78], SAFELI [23], Phan [56], SQLInjectionGen [38]</td>
</tr>
<tr>
<td>Black box testing</td>
<td>WAVES [31], SecuBat [39], Viper [12]</td>
</tr>
<tr>
<td>White box testing</td>
<td>MUSIC [64], Ardilla [42]</td>
</tr>
</tbody>
</table>
8.3.3 Dynamic analysis

Dynamic analysis techniques analyze the information acquired during program execution to detect SQL injection vulnerabilities. The information might be request and response patterns, structure of queries. Dynamic analysis can be performed at testing time during development or at run time after release. The drawback of dynamic analysis is that it only detects the vulnerabilities in the execution paths but it cannot detect which were not executed in the code [66].

8.3.4 Hybrid analysis

Hybrid analysis uses combination of both static and dynamic analysis to analyze the information obtained during program execution. Some techniques have used hybrid analysis to reduce the performance overhead and increase the efficiency to detect vulnerabilities. AMENSIA [26] uses the combination of static and dynamic analysis to counter against SQL injection attack. In the static analysis part, it analyzes the application code and automatically generates the model of legitimate queries. While in the dynamic part, it monitors all the dynamically generated queries at run time and compares with the legitimate queries built with static model.

8.3.5 Black box testing

Black box testing is a test design methods to detect vulnerabilities by testing application based on requirement specification. Requirement specification means what are the available inputs and the expected outputs that should result from each input. It is not concerned with application source code. For example, automated scanning techniques belongs to this category such as WAVES [31], SecuBat [39]. It is primarily used during testing time of the application.
8.3.6 White box testing

White box testing is also a test design methods to detect vulnerabilities by testing application with test cases. Test cases are generated from the internal structure of the system i.e. source code. For example, Ardilla[42] is a white box testing tool.

8.4 Classification by detection time

SQLIAs and their vulnerabilities can be detected at coding time, testing time or operating time (i.e. during operation).

8.4.1 Coding time

If SQLIA vulnerabilities are detected during coding time of an application development cycle, then it is considered as coding time detection. Detecting vulnerabilities in this early stage helps in tumbling the cost caused by tardy detection. Static analysis techniques detect SQLIA vulnerabilities during coding time without the need of code execution [66].

8.4.2 Testing time

If SQLIAs and their vulnerabilities are detected during testing time of an application development cycle, then it is considered as testing time detection. The different testing approaches, such as Black-box testing and White-box testing can be used as analysis methods in testing time for detecting attacks and their vulnerabilities [66].

8.4.3 Operation time

If SQLIAs and their vulnerabilities are detected at run time in the real world field after product is released then it is considered as operation time detection. Run time defense techniques usually prevent SQLIAs by terminating the execution of attacks
or sanitizing them. However, in case of false positives, terminating the execution can lead significant inconvenience to users [66].

8.5 Classification by detection location

SQLIAs and their vulnerabilities can be detected at various locations of the system. In our survey, we have classified them into four categories: client-side application, client-side proxy, server-side application, server-side proxy.

8.5.1 Client-side application

Client-side application techniques detect SQLIAs by analyzing HTML pages. While client-side scripts are also analyzed by some techniques which are used for detecting both SQLIA and cross-site scripting.

8.5.2 Server-side application

Server-side application technique detects SQLIAs by analyzing server-side application written in programming and script languages.

8.5.3 Client-side proxy

Client-side proxy acts as a gateway or intermediate server between a user and a web server. It intercepts user’s requests and responses from web server in order to detect SQLIAs. After detecting malicious inputs, either it rejects the request or alters malicious inputs to benign inputs.

8.5.4 Server-side proxy

Server-side proxy acts as supplementary server between an application server and a database server. It intercepts SQL queries from an application before reaching to
database server. It aids in blocking the malicious query execution in database.

8.6 Classification by response

Whenever the techniques detect the attacks, it responds to it. We classified five different categories based on the reaction when the SQL injection vulnerabilities are detected.

8.6.1 Report

Some of the defense techniques report whenever it detects vulnerabilities in the application. The report often consists of the vulnerable line number of the source code in the application. Static analysis and vulnerability testing tools generates reports.

8.6.2 Reject

Some of the defense techniques reject the user requests whenever it figures out that the user input is malicious and blocks the execution.

8.6.3 Escape

Some of the defense techniques instead of rejecting the user requests, tries to sanitize by escaping the malicious input. However, escaping malicious input is still vulnerable to SQL second order injection attacks.

8.6.4 User defined action

Application developer defines the action whenever they detect malicious input. They can set rules which will escape or encode the user input when a malicious pattern is found. It can be rejecting or escaping.
8.6.5 Code suggestion

Some techniques collect information from source code containing SQL Injection vulnerabilities and generate replacement secure code that can maintain applications functional integrity. It suggests the secure code whenever it detects vulnerability.

8.7 Classification by implementation

To deploy any techniques, developer needs to know if they require modifying the source code of the application. According to the implementation of the techniques, we have identified following two categories.

8.7.1 Modification of code base

The developer need to modify the source code of the application to deploy the SQL injection defense techniques. Therefore, it is often laborious, time consuming and tedious.

8.7.2 No modification of code base

The developers do not need to modify the source code of the application to deploy the SQL injection defense techniques. It provides flexibility and takes less effort in the implementation of the techniques.
Chapter 9

Evaluation Criteria

In our survey, we have identified three important characteristics which may have impact on the deployment of these defense techniques; accuracy, performance overhead and additional infrastructure. These evaluation criteria will provide a foundation to classify and compare the techniques.

9.1 Accuracy

Accuracy can be measured in terms of false positive and false negative generation. False positive occurs when the technique blocks the execution of non malicious input. False negative occurs when the technique allows actual malicious input. In our survey, most of the papers did not mention accuracy in detail. According to the level of detail provided by authors about false positives and false negatives, we classify them into three groups; concrete, explicit and implicit.

9.1.1 Concrete

The papers that demonstrate false positives and false negatives with evaluation results providing concrete values such as numbers or percentage, falls into this group.

9.1.2 Explicit

The papers that explicitly discuss the false positive and false negative without any evaluation result falls in this group.
9.1.3 Implicit

The papers that do not discuss any false positive and false negative, falls in this group. Although we can conclude that may cause false positives and false negatives from the information they have provided along with the technique.

Discussion of accuracy: There are several factors which may lead to false positive and false negatives. Grammar based approach generally considers user input containing SQL keywords as malicious. However, if any application accepts SQL keywords as a part of user input, it can generate false positive. Input signature approach can generate false positive if the input signature patterns are too restrictive. Output signature approach can generate false positive if output from execution includes predefined keywords such as error message where there is no such error. Moreover, incorrect specification of trusted input source or strings, string pattern matching and lack of essential policy can lead to false positives [66] as well. Tainted data flow approach can generate false positive when the list of distrusted source are not complete. Anomaly based approach can generate false positive when the training data set is not adequate to track normal behaviour. New vulnerability patterns, incorrect policy descriptions, incomplete list of vulnerable patterns can lead to false negatives.

9.2 Performance overhead

The execution time of any application is very crucial in today’s busy world. End users may get negative impact if they face any delay in the execution of application. Actually they may not want to compromise their valuable time with security measures. The overhead caused on the execution of a web application by the deployment of a defense technique is known as performance overhead. In other word, impact on overall performance of application due to the additional execution time of a defen-
sive technique itself is considered as performance overhead. In our survey, most of the papers have not discussed details about performance overhead. According to the level of detail provided by authors about performance overhead, we classify them into three groups; concrete, explicit and implicit where their definition are same as defined in above accuracy section.

9.3 Additional infrastructure

Defense techniques should be effortlessly adopted by developers and organizations. In order to successfully deploy the technique, other infrastructures (not including the technique itself) may be needed. Ease of use of any technique is very much correlated to the additional operational complexity required to deploy that technique. It can be classified into two categories.

9.3.1 Required

The technique which requires any kind of additional infrastructures to deploy falls into this category. For example, SQLGuard [10], SecurePHP [17], Prepared Statement [74], SQL DOM [52] etc require developer training. Developer training can be about training how to use secure libraries or APIs. Swaddler [14], Sdriver [55], Evolutionary Approach* [67], Anomaly detection* [76], etc require data training. Data is trained in the learning/training phase of anomaly detection.

9.3.2 Not required

The technique which does not require any additional infrastructure to deploy falls into this category. For example, AMNESIA [26], WASP [28], SAFELI [23], SANIA [44], etc does not require any additional infrastructure in order to deploy.
Chapter 10

Data Analysis

In this chapter, we have presented the analysis in the form of description based on data as provided by authors in the papers. Further, we are going to talk about the status of SQLIA, trends and accuracy related to the SQL injection defense techniques. We also discussed the correlation between the type of detection principles and accuracy of the techniques.

10.1 Status of SQLIA

According to the Open Web Application Security Project (OWASP), many web applications are still vulnerable to SQLIAs [60]. In fact, SQL injection occupied the top spot in OWASP top ten list for 2010. Every three year, OWASP releases top ten lists of most dangerous security flaws in web applications. Figure 1.1 depicted in the above introduction chapter, shows the ranking of SQL injection according to the OWASP top ten lists over the years. In 2004, it occupied sixth position. In 2007, it was in the second position but now in 2010, it emerged in the first position. This data shows the urgency to protect the web application against the SQL injection attacks.

10.2 Trends

A lot of research has been done and many defense techniques have been proposed. The number of the techniques has been increasing with some downfall since the year
2002 until 2010 as shown in figure 10.1. It depicts the number of defense techniques over the years. We can clearly see an increasing amount of work being done over the years.

![Number of defense techniques over the years](image)

**Figure 10.1: Number of defense techniques over the years**

The proposed techniques employed different detection principles to defend against SQLIAs. Figure 10.2 shows the percentage of different detection principles used.

Of the total, 49% use grammar based violation, 25% use signature based, and 13% each for both tainted data flow and anomaly based. The detection principles were not spread evenly over the years as shown in figure 10.3. Figure 10.3 depicts the usage of different detection principles over the years.

We have encountered different analysis methods used in the techniques to detect SQL injection vulnerabilities. Figure 10.4 shows the percentage of different analysis methods used. Most of the techniques have used dynamic analysis whereas white box testing was least used analysis method. Hybrid analysis which is the combination of static and dynamic analysis was second mostly used one. Static analysis and secure programming have been used equally.

We found that these analysis methods were not evenly used over the years. Figure
Figure 10.2: Usage of detection principles

Figure 10.3: Usage of detection principles over the years
10.5 depicts the usage of analysis methods over the years.

In our survey, we have identified three important characteristics that have impact on the deployment of these defensive techniques; accuracy, performance overhead and additional infrastructure. Only 22 techniques have explained the accuracy with concrete statistical results in terms of false positives and only 15 techniques in terms of false negatives while the rest, did not demonstrate any statistical results. Only 17 techniques have presented their concrete statistical performance overhead results while the rest have explained the performance overhead either explicitly or implicitly without any statistical data. Some techniques require additional infrastructures to deploy the techniques successfully. 31 techniques require additional infrastructures. Figure 10.6 depicts the number of techniques that provides different level of details (implicit, explicit or concrete) in terms of false positive, false negative, and performance overhead.
Figure 10.5: Usage of analysis methods over the year

Figure 10.6: Level of details in terms of false positive, false negative and performance overhead
10.3 Discussion of false positive and false negative

In this section, we presented the information about actual false positive and negative data as reported by authors of the techniques. Some authors have discussed what they have done to minimize the false positive and negative rates. We have provided those information in brief as well. Since all the authors have not reported the concrete data of accuracy, we could not discuss the techniques which lacked concrete data. We tried to discuss about some of the techniques which have presented explicit data as well.

A 100% accuracy is claimed by some techniques such as SQLrand [8], AMNESIA [26], SQLGuard [10], SQLCHECK [68], CANDID [6], APPASIA* [54], WASP [28], SQL-IDS [40], ABCDM* [72], ADRILLA [42], SQLProb [48].

Authors have evaluated SQLrand [8] proxy on the open source application such as phpBB v2.0.5 (PHP bulletin board) and Php-Nuke (content management system) and argued that using SQLrand proxy provided 100% accuracy. However, they have not provided any empirical data to support their claim in the paper.

The empirical evaluation of AMNESIA [26] was performed on seven web applications of various types and sizes and it was able to stop all of the 1,470 attacks without producing any false positive for the 3,500 legitimate queries to the applications i.e. without zero false negatives. The seven web applications were Checkers, Office talk, Employee directory, Bookstore, Events, Classifieds and Portal. Authors have demonstrated detail empirical results i.e. number of unsuccessful attacks, the number of successful attacks, and the number of attacks detected and reported.

The authors of SQLGuard [10] had performed the case study on Geological Hazard Inventory Management System which was built on J2EE using JSP with Apache Tomcat as the application server. Authors argued that SQLGuard handled every attempted injection with zero false positives and negatives. The empirical data of the experiment was not demonstrated in the paper.
Authors have evaluated SQLCHECK [68] on five real world web applications developed in PHP and JSP with real world attacks and legitimate data as input provided by an independent research group. SQLCHECK prevented all attacks i.e. zero false negatives and allowed all legitimate uses i.e. no false positives. The five real world applications were employee directory, events, classifieds, portal and bookstore.

CANDID [6] was evaluated using AMNESIA test bed which contained both attack and non attack benign inputs. An attack input was based on many different vectors of SQL code injections. A CANDID instrumented application was tested with the attack suite. It was able to defend all the attacks, and there were no false positives. Authors have presented the detailed result which consist the number of input attempts, the number of successful attacks, the number of attacks detected and the number of non attack benign inputs.

An old SQL-injection prone version of phpBB (version 2.0.0) was used to evaluate the APPASIA* [54]. 176 attacks were possible in the old version. Authors have implemented APPASIA* on the old version and newly protected version was evaluated again. The experiment showed zero false positive and false negative.

WASP [28] was tested on ten web applications. Authors have developed the attack sets with the help of master level student which have expertise on developing penetration testing tools. The result of the attack set contains 26 unique attacks. All type of attacks except second order injection (multiphase attacks) was represented in the set. Second order injection attack requires human intervention therefore authors have omitted those to make test bed fully automatic. WASP was able to identify all SQLIA without generating any false positives. WASP was able to identify 12,826 viable SQLIAs without stopping any of the 13,166 legitimate accesses. WASP have 0 % false positive and 0 % false negative. Legitimate access represents for false positive and total attacks detected represents for false negative.

To evaluate SQL-IDS [40], authors have performed an experiment on book store
web application. The attacks set consists of 2,450 SQL queries, out of which 420 were poisoned with SQL injection attacks and 2,030 were attack free SQL statements. The first experiment showed that all 420 attacks were detected by the tool showing 100% effectiveness i.e. zero false negatives. Another experiment to evaluate tools precision showed that all 2030 attack free SQL queries were allowed i.e. zero false positives.

In this paper ABCDM* [72], authors have evaluated two character distribution models; the FCD (frequency character distribution) and SCC (same character comparison) models and showed that SCC model is very effective and more accurate than FCD at detecting SQL injection attacks. Authors mentioned that the requirement of the training dataset which will contain all possible SQL queries would be difficult to generate in practice. The experiments were performed in the phpBB (bulletin board) application and phpNuke (content management system) open source application written in PHP. Vulnerable Versions of phpBB (v 2.0.0 and v 2.5.0) and phpNuke (v 6.0, v 7.3 and v7.5), running on an Apache v1.3.34 web server using PHP v4.4.2 and MySQL v4.1.11 were utilized. Authors have presented the detection and false positive results for UNION and Tautology attacks for both the FCD and SCC models on the phpBB application. The FCD model produced zero false positives for all four datasets (smilies, words, VT05 and VT06) in the phpBB application when testing for UNION attacks. The results of the experiment showed that the SCC model detected all UNION attacks across all four datasets with zero false positive rates.

Authors have evaluated Ardilla [42] on five PHP applications. Authors have found 68 previously unknown vulnerabilities which contain 23 SQLI, 33 first-order XSS, and 12 second-order XSS. Authors inspected all 23 SQLI reports and found no false positives. The five PHP application were Schoolmate 1.5.4 (tool for school administration), Webchess 0.9.0 (online chess game), FAQforge 1.3.2 (tool for creating and managing documents), Eve 1.0 (player activity tracker for an online game), and
Gecchblite 0.1 (a simple bulletin board).

The authors of SQLProb [48] have used the same AMNESIA attack test suite containing both benign and real attack patterns. Authors have further extended the attack patterns to ensure the completeness of attack suite by including a wide range of the real-world attacking patterns. Authors ran the attack suite and claimed that SQLProb was able to correctly identify all the attacks in the test suite showing 100% accuracy. The empirical data of test result was not presented in the paper.

The authors of WAVES [31] have reported about false negativity in their tool and they proposed the use of ”deep injection” mechanism to eliminate certain types of false negatives. But authors have not mentioned any the false positive data in the paper.

In WEBSARRI [32], the authors have shown that when tested, their tool yielded a false positive rate of 26.9 percent. Authors have discussed that if variable labels become too restrictive during computation, it will result in high false positive rates. False positives may arise from the imprecise approximation of temporal variable properties and runtime information manipulation or validation. False positive rates can be effectively reduced by making more specific approximations of the run time information flow and automated declassification. After adding support for type-aware qualifiers, false positive rate of WEBSARRI was reduced by 10.03 percent.

Authors of Java Static Tainting [49] have presented a detailed experimental evaluation on nine open-source Java applications. Authors have reported a total of 29 security vulnerabilities, and analysis reports only 12 false positives in the application. Imprecise object naming was the main cause for 12 false positives. The nine web applications were Jboard, Blueblog, Webgoat, Blojsom, Personalblog, Snipsnap, Road2hibernate, Pebble and Roller. According to the authors, combination of context sensitivity and improved object naming proved instrumental in reducing the number of false positives. After combining context sensitivity and improved object naming, the number of false positives reduces to zero for all applications except

50
Snipsnap. Authors have eliminated all the false positives in Snipsnap by creating a new object name at every call to, StringWriter.toString().

Authors have evaluated the false positive rate of Anomaly Detection* [76] using an installation of the PHP-Nuke web portal system. Authors have used three attack-free datasets; one for training the models, the second for the threshold learning phase, and the third for false positive rate estimation. The results of the test showed the high rate of false positive. The reason behind the high false alarm rate is due to the training data which were generated in a different month than the test data. Authors have changed the configuration by introducing two custom data types: month and year. They re-evaluated the system and found out the false positive rate was dramatically reduced. The total number of Attack free Queries used was 15704. After re-evaluation, false positives was reduced from $58(0.37\%)$ to $2(0.013\%)$.

According to the authors of ESIATDM* [58], techniques that employ input validation are most prone to a great number of false positives, however false negatives also cannot be ruled out. Authors have tested different types of SQLIAs namely; tautology, access control violation, inserting additional SQL statements and second order injection on both protected and unprotected web servers. Authors claimed that the technique has detected all types of SQLIAs without any empirical results.

Authors of CSSE [59] showed that their technique may generate false positive and false negative in some situation. They described three such situations, namely; incomplete implementations, incorrect implementations and invalid assumptions. There is no empirical result presented in the paper in order to evaluate the accuracy of the technique.

Authors have evaluated the effectiveness of SWADDLER [14] by training the system on the five different test applications namely; BloggIt, PunBB, Scarf, SimpleCms and WebCalendar. The experiment showed that SWADDLER produced a few false positives. Authors figured out the reason for the false alarms. It was caused by the execution of parts that were trained by a limited number of requests during
the training phase.

Authors of DMSQL* [20] claimed that it has 100% accuracy. There is no empirical data presented in the paper.

SANIA [44], an SQLIA vulnerability testing tool was evaluated using six real-world web applications. SANIA was able to identify 39 vulnerabilities and causing only 13 false positives (about 30%). To reduce the false positive, SANIA allows the application developers to be involved so as to control the matching of parse trees in the SQLIA vulnerability testing.

Authors have evaluated AMSMI* [47], a hybrid technique and found out the technique raised some false positives (about 40%) and false negatives (about 7%). Authors have argued that blacklist as input validation may induces more false negatives while whitelist may induce more false positives. Authors demonstrated using hybrid method i.e. using whitelist, blacklist and the special case as input validation will result in a few false positives and negatives.

Authors have performed case studies on two small web applications (Book store and cabinet applications) to evaluate SQLInjectionGen [38]. It had zero false positives, but had a small number of false negatives. They compared the results with FINDBUGs (commercial tool to detect bugs in java program) and showed the promising results.

Authors have performed experiments to evaluate Evolutionary Approach* [67]. The average false positive and negative found about 23%.

Authors have observed that IAAT* [4] was able to detect 21 vulnerabilities as compared to 7 by a freeware tool Paros (a web application scanner). IAAT* had 16 false positives. False positives were due to the length of the attack code and mistaken backslashes.
10.4 Correlation analysis

In this section, we discuss about the correlation between some of the important variables. Correlation analysis is very valuable for the formation of hypothesis.

We have analysed all the available data and performed pair wise correlation. We utilized the concrete data provided by authors to calculate Pearson Correlation Coefficient. We are showing only those correlations that were found statistically significant. The correlation between detection principles and accuracy of the techniques was one of it. Pearson Correlation Coefficient between grammar based violation detection principle and accuracy was calculated 0.970. The result implies large positive correlation between those two parameters.

The correlation coefficient between tainted data flow detection principle and accuracy was 0.949 which is also large positive correlation. It means that both variables move in the same direction together. Due to lack of concrete data, we could not correlate the signature based detection principle and anomaly detection principle with accuracy (false positive and false negative) of the techniques. With the basis of this statistical measurement, we can hypothesize that the techniques which employ grammar based violation detection principle have higher rate of accuracy as compared to other detection principles. Due to the lack of concrete data, we could not correlate other important parameters. Therefore, many other things could also be true beside above hypothesis.
Chapter 11

Conclusion and Future Work

The SQL injection field contains numerous defense techniques, which obscure a global view of the SQL injection attack problem. We have conducted a systematic literature survey and presented a taxonomy of SQL injection defense techniques which can be seen as a first attempt to cut through the obscurity. The proposed taxonomy insights into the field of SQL injection and provides a number of directions for future research.

We have performed a systematic review for the survey and reviewed 61 different SQL injection defense techniques and presented classification of these techniques according to the taxonomy. Our taxonomy consists of various dimensions which can be grouped under two higher order terms; detection method and evaluation criteria. Detection method consists of seven dimensions; nature of defense, detection principle, analysis method, detection time, detection location, response and implementation while evaluation criteria consists of three dimensions; accuracy (false positive and false negative), performance overhead and additional infrastructure. As different detection principles and analysis methods have different efficiency and coverage of vulnerability detection, our taxonomy helps the organization(s) to understand these differences. Hence, on the basis of these differences, organization(s) can choose appropriate technique(s) depending on their available resources and environments.

As a matter of fact, a complete and final taxonomy cannot be built by reviewing only 61 techniques because this research area is immense in dimension. Nevertheless, we tried our best to make our taxonomy more diverse. Nonetheless, there may be rooms for additional sub-categories. For instance, additional infrastructure can
be further classified into more sub-categories, though at the moment, we have classified it only into twofold. Our future work will be to extend it. Due to lack of enough statistical data provided by authors, we could not perform more pair-wise correlation. As we have already mentioned that, different authors have presented their work at different levels of detail, extracting uniform data from such a diverse range of papers was a very tedious task. This is the reason as to why we have made certain assumptions and calculations in some places, based on the information and knowledge we have. A standard measurement and presentation of the evaluation criteria will be our another future work.
References


M. S. SAM. Sql injection protection by variable normalization of sql statement, 2005.


Y. Shin and L. Williams. Toward a taxonomy of techniques to detect cross-site scripting and sql injection vulnerabilities. 2007.


Appendix A

Taxonomy categorizations of techniques according to detection method

This appendix provides a table of defense techniques with categories of detection method according to our taxonomy described in this paper.
<table>
<thead>
<tr>
<th>Technique</th>
<th>Nature of defense</th>
<th>Detection principle</th>
<th>Analysis method</th>
<th>Detection time</th>
<th>Detection location</th>
<th>Response</th>
<th>Modification of code base</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security Gateway [63]</td>
<td>Countermeasure</td>
<td>Input Signature</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Server side proxy</td>
<td>User defined action</td>
<td>No</td>
</tr>
<tr>
<td>WAVES [31]</td>
<td>Detection</td>
<td>Output Signature</td>
<td>Black box testing + Dynamic</td>
<td>Testing time</td>
<td>Client side application</td>
<td>Report</td>
<td>No</td>
</tr>
<tr>
<td>AUSELSQI [1]</td>
<td>Countermeasure</td>
<td>Input Signature</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Client side proxy</td>
<td>Reject</td>
<td>No</td>
</tr>
<tr>
<td>SQLrand [8]</td>
<td>Countermeasure</td>
<td>Grammar based violation</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Server side proxy</td>
<td>Reject</td>
<td>Yes</td>
</tr>
<tr>
<td>WebSSARI [32]</td>
<td>Prevention</td>
<td>Negative tainted data flow</td>
<td>Hybrid</td>
<td>Run time</td>
<td>Server side application</td>
<td>Code suggestion</td>
<td>No</td>
</tr>
<tr>
<td>Tautology checker* [77]</td>
<td>Prevention</td>
<td>Grammar based violation</td>
<td>Static</td>
<td>Run time</td>
<td>Server side application</td>
<td>Code suggestion</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Countermeasure</td>
<td>Grammar based violation</td>
<td>Dynamic</td>
<td>Runtime</td>
<td>Server side application</td>
<td>User defined action</td>
<td>Yes/No</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------------</td>
<td>------------------------</td>
<td>---------</td>
<td>---------</td>
<td>--------------------------</td>
<td>--------------------</td>
<td>--------</td>
</tr>
<tr>
<td>SQLGuard [10]</td>
<td>Countermeasure</td>
<td>Grammar based violation</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Server side application</td>
<td>User defined action</td>
<td>Yes</td>
</tr>
<tr>
<td>Variable Normalization [62]</td>
<td>Countermeasure</td>
<td>Input Signature</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Server side proxy</td>
<td>Reject</td>
<td>No</td>
</tr>
<tr>
<td>Java Dynamic Tainting [25]</td>
<td>Countermeasure</td>
<td>Negative tainted data flow</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Server side application</td>
<td>Reject</td>
<td>No</td>
</tr>
<tr>
<td>AMNESIA [26]</td>
<td>Countermeasure</td>
<td>Grammar based violation</td>
<td>Hybrid</td>
<td>Run time</td>
<td>Server side application</td>
<td>Reject + Report</td>
<td>No</td>
</tr>
<tr>
<td>Java Static Tainting [49]</td>
<td>Prevention</td>
<td>Negative tainted data flow</td>
<td>Static</td>
<td>Run time</td>
<td>Server side application</td>
<td>Report + Code suggestion</td>
<td>No</td>
</tr>
<tr>
<td>SecuriFly [51]</td>
<td>Countermeasure</td>
<td>Negative tainted data flow</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Server side application</td>
<td>User defined action</td>
<td>No</td>
</tr>
<tr>
<td>SQL DOM [52]</td>
<td>Countermeasure</td>
<td>Input Signature</td>
<td>Secure Programming</td>
<td>Compile time</td>
<td>Server side application</td>
<td>Escape</td>
<td>Yes</td>
</tr>
<tr>
<td>Anomaly Detection* [76]</td>
<td>Countermeasure</td>
<td>Learning based Anomaly Detection</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Server side application</td>
<td>Reject</td>
<td>No</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------</td>
<td>---------------------------------</td>
<td>---------</td>
<td>----------</td>
<td>--------------------------</td>
<td>--------</td>
<td>----</td>
</tr>
<tr>
<td>Pixy [37]</td>
<td>Prevention</td>
<td>Negative tainted data flow</td>
<td>Static</td>
<td>Coding time</td>
<td>Server side application</td>
<td>Report</td>
<td>No</td>
</tr>
<tr>
<td>SecuBat [39]</td>
<td>Detection</td>
<td>Output Signature</td>
<td>Black box testing</td>
<td>Testing time</td>
<td>Client side application</td>
<td>Report</td>
<td>No</td>
</tr>
<tr>
<td>AntiMaliciousInjection [46]</td>
<td>Countermeasure</td>
<td>Input Signature</td>
<td>Secure Programming</td>
<td>Coding time</td>
<td>Server side application</td>
<td>Escape + Reject</td>
<td>No</td>
</tr>
<tr>
<td>ESIATDM* [58]</td>
<td>Countermeasure</td>
<td>Grammar based violation</td>
<td>Hybrid</td>
<td>Run time</td>
<td>Server side application</td>
<td>Reject</td>
<td>No</td>
</tr>
<tr>
<td>CSSE [59]</td>
<td>Countermeasure</td>
<td>Grammar based violation</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Server side application</td>
<td>Reject</td>
<td>No</td>
</tr>
<tr>
<td>SQLCHECK [68]</td>
<td>Countermeasure</td>
<td>Grammar based violation</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Server side proxy</td>
<td>Reject</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Countermeasure</td>
<td>Grammar based violation</td>
<td>Hybrid</td>
<td>Run time</td>
<td>Server side proxy</td>
<td>Reject + Report</td>
<td>No</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------</td>
<td>-------------------------</td>
<td>--------</td>
<td>-----------</td>
<td>-------------------</td>
<td>----------------</td>
<td>-----</td>
</tr>
<tr>
<td>StoredProcedure* [78]</td>
<td>Countermeasure</td>
<td>Grammar based violation</td>
<td>Hybrid</td>
<td>Run time</td>
<td>Server side proxy</td>
<td>Reject + Report</td>
<td>No</td>
</tr>
<tr>
<td>Swaddler [14]</td>
<td>Detection</td>
<td>Learning based Anomaly Detection</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Server side application</td>
<td>Reject</td>
<td>No</td>
</tr>
<tr>
<td>DMSQL* [20]</td>
<td>Countermeasure</td>
<td>Learning based Anomaly Detection</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Server side application</td>
<td>Reject</td>
<td>No</td>
</tr>
<tr>
<td>AProSec [30]</td>
<td>Detection</td>
<td>Input Signature</td>
<td>Dynamic</td>
<td>Compile time</td>
<td>Client + Server side proxy</td>
<td>Escape</td>
<td>No</td>
</tr>
<tr>
<td>Smask [36]</td>
<td>Countermeasure</td>
<td>Grammar based violation</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Server side application</td>
<td>Escape</td>
<td>Yes</td>
</tr>
<tr>
<td>APPASIA* [54]</td>
<td>Countermeasure</td>
<td>Grammar based violation</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Server side application</td>
<td>Reject</td>
<td>No</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Prevention Method</td>
<td>Grammar Based Violation</td>
<td>Secure Programming</td>
<td>Run time</td>
<td>Server side application</td>
<td>Code suggestion</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
<td>----------------------------------------</td>
<td>-------------------------</td>
<td>--------------------</td>
<td>----------</td>
<td>------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>AFGSSS* [73]</td>
<td>Prevention</td>
<td></td>
<td>Grammar based violation</td>
<td>Secure Programming</td>
<td>Run time</td>
<td>Server side application</td>
<td>Code suggestion</td>
</tr>
<tr>
<td>SAFELI [23]</td>
<td>Countermeasure</td>
<td></td>
<td>Grammar based violation</td>
<td>Hybrid</td>
<td>Compile time</td>
<td>Server side application</td>
<td>Escape</td>
</tr>
<tr>
<td>WASP [28]</td>
<td>Countermeasure</td>
<td></td>
<td>Positive tainted data flow + grammar based violation</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Server side application</td>
<td>User defined action</td>
</tr>
<tr>
<td>SQL-IDS [40]</td>
<td>Countermeasure</td>
<td></td>
<td>Grammar based violation</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Server side proxy</td>
<td>Reject + Report</td>
</tr>
<tr>
<td>ABCDM* [41]</td>
<td>Countermeasure</td>
<td></td>
<td>Learning based Anomaly detection</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Client side proxy</td>
<td>Reject</td>
</tr>
<tr>
<td>AMSMI* [47]</td>
<td>Countermeasure</td>
<td></td>
<td>Input Signature</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Server side proxy</td>
<td>Reject</td>
</tr>
<tr>
<td>Sdriver [55]</td>
<td>Countermeasure</td>
<td>Output Signature</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Server side proxy</td>
<td>Reject</td>
<td>No</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------</td>
<td>------------------</td>
<td>---------</td>
<td>----------</td>
<td>--------------------</td>
<td>--------</td>
<td>----</td>
</tr>
<tr>
<td>MUSIC [64]</td>
<td>Detection</td>
<td>Learning based anomaly Detection</td>
<td>White box testing + Dynamic</td>
<td>Testing time</td>
<td>Server side application</td>
<td>Report</td>
<td>No</td>
</tr>
<tr>
<td>CVSID [80]</td>
<td>Detection</td>
<td>Grammar based violation</td>
<td>Static</td>
<td>Testing time</td>
<td>Server side application</td>
<td>Report</td>
<td>No</td>
</tr>
<tr>
<td>CAPSIA* [18]</td>
<td>Countermeasure</td>
<td>Input Signature</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Server side application</td>
<td>Reject</td>
<td>No</td>
</tr>
<tr>
<td>WBSCA* [19]</td>
<td>Countermeasure</td>
<td>Learning based anomaly Detection</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Server side application</td>
<td>Reject + Report</td>
<td>No</td>
</tr>
<tr>
<td>SQLInjectionGen [38]</td>
<td>Detection</td>
<td>Grammar based violation</td>
<td>Hybrid</td>
<td>Testing time</td>
<td>Server side application</td>
<td>Report</td>
<td>Yes</td>
</tr>
<tr>
<td>Countermeasure</td>
<td>Detection</td>
<td>Negative tainted data flow</td>
<td>White box testing</td>
<td>Testing time</td>
<td>Server side application</td>
<td>Report</td>
<td>No</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------</td>
<td>----------------------------</td>
<td>-------------------</td>
<td>--------------</td>
<td>-------------------------</td>
<td>--------</td>
<td>----</td>
</tr>
<tr>
<td>SQLProb [48]</td>
<td>Countermeasure</td>
<td>Grammar based violation</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Server side proxy</td>
<td>Reject</td>
<td>No</td>
</tr>
<tr>
<td>ADMIRE [50]</td>
<td>Prevention</td>
<td>Grammar based violation</td>
<td>Static</td>
<td>Coding time</td>
<td>Server side application</td>
<td>Report</td>
<td>No</td>
</tr>
<tr>
<td>Phan [56]</td>
<td>Countermeasure</td>
<td>Negative tainted data flow</td>
<td>Hybrid</td>
<td>Run time</td>
<td>Server side application</td>
<td>User defined action</td>
<td>No</td>
</tr>
<tr>
<td>MDWAA* [61]</td>
<td>Prevention</td>
<td>Input Signature + Programmed based anomaly detection</td>
<td>Dynamic</td>
<td>Testing time</td>
<td>Server side application</td>
<td>Report</td>
<td>Yes</td>
</tr>
<tr>
<td>SBSQLID [65]</td>
<td>Countermeasure</td>
<td>Grammar based violation + Input Signature</td>
<td>Dynamic Runtime</td>
<td>Server side application</td>
<td>Reject</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Evolutionary Approach* [67]</td>
<td>Countermeasure</td>
<td>Programmed based anomaly Detection</td>
<td>Dynamic Runtime</td>
<td>Server side application</td>
<td>Reject</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>CANDID [6]</td>
<td>Countermeasure</td>
<td>Grammar based violation</td>
<td>Dynamic Runtime</td>
<td>Server side application</td>
<td>Reject</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>TAPS [7]</td>
<td>Countermeasure</td>
<td>Grammar based violation</td>
<td>Dynamic Runtime</td>
<td>Server side application</td>
<td>Reject</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Countermeasure</td>
<td>Grammar based violation</td>
<td>Secure Programming</td>
<td>Coding time</td>
<td>Server side application</td>
<td>Reject</td>
<td>Report</td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td>-------------------------</td>
<td>--------------------</td>
<td>-------------</td>
<td>-------------------------</td>
<td>--------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>Syntax Embeddings* [9]</td>
<td></td>
<td>Secure Programming</td>
<td>Coding time</td>
<td>Server side application</td>
<td>Reject</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Viper [12]</td>
<td>Detection</td>
<td>Output Signature</td>
<td>Black box testing</td>
<td>Client side application</td>
<td>Report</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>DUD [15]</td>
<td>Countermeasure</td>
<td>Grammar based violation</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Server side application</td>
<td>Reject</td>
<td>No</td>
</tr>
<tr>
<td>ACMWDPC* [35]</td>
<td>Countermeasure</td>
<td>Positive tainted date flow</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Server side application</td>
<td>Reject</td>
<td>No</td>
</tr>
<tr>
<td>Query Parser [45]</td>
<td>Countermeasure</td>
<td>Grammar based violation</td>
<td>Dynamic</td>
<td>Run time</td>
<td>Server side application</td>
<td>Report</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Countermeasure</td>
<td>Grammar based violation</td>
<td>Dynamic Runtime</td>
<td>Clientside proxy</td>
<td>Reject</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------</td>
<td>-------------------------</td>
<td>-----------------</td>
<td>------------------</td>
<td>--------</td>
<td>----</td>
<td></td>
</tr>
<tr>
<td>ANWAF* [57]</td>
<td></td>
<td></td>
<td>Dynamic</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D-WAV [79]</td>
<td>Prevention</td>
<td>Grammar based violation</td>
<td>Dynamic</td>
<td>Testing time</td>
<td>Report</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>X LOG AUTHENTICATION* [33]</td>
<td>Countermeasure</td>
<td>Grammar based violation + Input Signature</td>
<td>Dynamic</td>
<td>Run time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix B

Taxonomy categorizations of techniques according to evaluation criteria

This appendix provides a table of defense techniques with categories of evaluation criteria according to our taxonomy described in this paper. In Table 2, FP means false positive and FN means false negative.
Table B.1: Summary of classification of defense techniques according to evaluation criteria

<table>
<thead>
<tr>
<th>Techniques</th>
<th>Accuracy- in terms of FP and FN</th>
<th>Performance overhead</th>
<th>Addition infrastructure</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAVES [31]</td>
<td>FP: implicit, FN: concrete</td>
<td>concrete</td>
<td>required</td>
</tr>
<tr>
<td>SQLRand [8]</td>
<td>FP: implicit, FN: implicit</td>
<td>concrete</td>
<td>not required</td>
</tr>
<tr>
<td>Tautology checker* [77]</td>
<td>FP: implicit, FN: implicit</td>
<td>implicit</td>
<td>not required</td>
</tr>
<tr>
<td>Techniques</td>
<td>Accuracy- in terms of FP and FN</td>
<td>Performance overhead</td>
<td>Addition infrastructure</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------------</td>
<td>----------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>SecuriFly [51]</td>
<td>FP: explicit, FN: explicit</td>
<td>concrete</td>
<td>not required</td>
</tr>
<tr>
<td>SQL DOM [52]</td>
<td>FP: implicit, FN: implicit</td>
<td>concrete</td>
<td>required</td>
</tr>
<tr>
<td>Anomaly Detection* [76]</td>
<td>FP: concrete, FN: implicit</td>
<td>concrete</td>
<td>required</td>
</tr>
<tr>
<td>Pixy [37]</td>
<td>FP: implicit, FN: implicit</td>
<td>implicit</td>
<td>required</td>
</tr>
<tr>
<td>AntiMaliciousInjection [46]</td>
<td>FP: implicit, FN: implicit</td>
<td>implicit</td>
<td>required</td>
</tr>
<tr>
<td>ESIATDM* [58]</td>
<td>FP: concrete, FN: concrete</td>
<td>concrete</td>
<td>required</td>
</tr>
<tr>
<td>CSSE [59]</td>
<td>FP: explicit, FN: explicit</td>
<td>concrete</td>
<td>required</td>
</tr>
<tr>
<td>Techniques</td>
<td>Accuracy- in terms of FP and FN</td>
<td>Performance overhead</td>
<td>Addition infrastructure</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------------------</td>
<td>----------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>StoredProcedure*</td>
<td>FP: explicit FN: implicit</td>
<td>concrete</td>
<td>not required</td>
</tr>
<tr>
<td>[78]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sania [44]</td>
<td>FP: concrete FN: implicit</td>
<td>implicit</td>
<td>not required</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>APPASIA* [54]</td>
<td>FP: concrete FN: concrete</td>
<td>implicit</td>
<td>not required</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AFGSSS* [73]</td>
<td>FP: explicit FN: explicit</td>
<td>explicit</td>
<td>required</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAFELI [23]</td>
<td>FP: implicit FN: implicit</td>
<td>implicit</td>
<td>not required</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL-IDS [40]</td>
<td>FP: concrete FN: concrete</td>
<td>concrete</td>
<td>not required</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

81
<table>
<thead>
<tr>
<th>Techniques</th>
<th>Accuracy- in terms of FP and FN</th>
<th>Performance overhead</th>
<th>Addition infrastructure</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCDM* [41]</td>
<td>FP: concrete FN: concrete</td>
<td>implicit</td>
<td>required</td>
</tr>
<tr>
<td>AMSMI* [47]</td>
<td>FP: concrete FN: concrete</td>
<td>implicit</td>
<td>required</td>
</tr>
<tr>
<td>MUSIC [64]</td>
<td>FP: implicit FN: implicit</td>
<td>implicit</td>
<td>required</td>
</tr>
<tr>
<td>CVSID [80]</td>
<td>FP: implicit FN: implicit</td>
<td>implicit</td>
<td>not required</td>
</tr>
<tr>
<td>CAPSIA* [18]</td>
<td>FP: implicit FN: implicit</td>
<td>explicit</td>
<td>not required</td>
</tr>
<tr>
<td>WBSCA* [19]</td>
<td>FP: concrete FN: explicit</td>
<td>implicit</td>
<td>required</td>
</tr>
<tr>
<td>SQLInjectionGen [38]</td>
<td>FP: concrete FN: concrete</td>
<td>implicit</td>
<td>not required</td>
</tr>
<tr>
<td>Ardilla [38]</td>
<td>FP: concrete FN: concrete</td>
<td>concrete</td>
<td>required</td>
</tr>
<tr>
<td>ADMIRE [50]</td>
<td>FP: implicit FN: implicit</td>
<td>implicit</td>
<td>not required</td>
</tr>
<tr>
<td>Techniques</td>
<td>Accuracy- in terms of FP and FN</td>
<td>Performance overhead</td>
<td>Addition infrastructure</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---------------------------------</td>
<td>----------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>Phan [56]</td>
<td>FP: implicit FN: implicit</td>
<td>concrete</td>
<td>not required</td>
</tr>
<tr>
<td>MDWAA* [61]</td>
<td>FP: concrete FN: concrete</td>
<td>concrete</td>
<td>not required</td>
</tr>
<tr>
<td>SBSQLID [65]</td>
<td>FP: implicit FN: implicit</td>
<td>implicit</td>
<td>required</td>
</tr>
<tr>
<td>Evolutionary Approach* [67]</td>
<td>FP: concrete FN: concrete</td>
<td>implicit</td>
<td>required</td>
</tr>
<tr>
<td>Techniques</td>
<td>Accuracy-in terms of FP and FN</td>
<td>Performance overhead</td>
<td>Addition infrastructure</td>
</tr>
<tr>
<td>----------------------------</td>
<td>--------------------------------</td>
<td>----------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>DUD [15]</td>
<td>FP: explicit</td>
<td>explicit</td>
<td>not required</td>
</tr>
<tr>
<td></td>
<td>FN: explicit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACMWDPC* [35]</td>
<td>FP: implicit</td>
<td>concrete</td>
<td>not required</td>
</tr>
<tr>
<td></td>
<td>FN: implicit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Query Parser [45]</td>
<td>FP: implicit</td>
<td>implicit</td>
<td>not required</td>
</tr>
<tr>
<td></td>
<td>FN: implicit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANNWAF* [57]</td>
<td>FP: explicit</td>
<td>explicit</td>
<td>required</td>
</tr>
<tr>
<td></td>
<td>FN: implicit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D-WAV [79]</td>
<td>FP: explicit</td>
<td>implicit</td>
<td>not required</td>
</tr>
<tr>
<td></td>
<td>FN: implicit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X LOG AUTHENTICATION* [33]</td>
<td>FP: explicit</td>
<td>concrete</td>
<td>not required</td>
</tr>
<tr>
<td></td>
<td>FN: explicit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix C

Summary of reviewed techniques

C.1 Abstracting Application-Level Web Security

Authors have presented a structuring technique in order to help developer abstract security policies from many web applications. Authors have claimed that technique will protect application level attacks. The techniques will help programmers to develop secure applications which are resilient to a wide range of common attacks. The system consists of three main components.

- Security policy description language (SPDL): Developer specifies a set of validation transformation rules in SPDL. The transformation rules describe what to do when a malicious pattern is found from user input.

- Policy compiler: Policy compiler translates these specifications into code for checking validation routines.

- Security gateway: Security gateway is placed between client and web server. It will act as application level firewall and dynamically intercept, analyze and transforms HTTP messages to enforce the specified policy.

Authors have presented concrete performance results in the paper.
C.2 Web Application Security Assessment by Fault Injection and Behavior Monitoring

This technique is not specific to only SQL injection attacks; it detects cross-site scripting attacks as well. It employs a web crawler to identify all SQLIA vulnerable points in a Web application. Then it builds attacks based on a particular list of attack techniques and patterns that targets such vulnerable points. Finally it monitors the response of application to the attacks and employs machine learning techniques to amend its attack methodology. Since it uses machine learning approach in testing, it is considered as improved one than other most penetration testing approaches. However, like all penetration testing and black-box testing techniques, it cannot present guarantees of completeness. The authors have implemented it in WAVES (Wave Application Vulnerability and Error Scanner) tool.

C.3 An Automated Universal Server Level Solution for SQL Injection Security Flaw

AUSELSQI is an SQL injection defense technique which safeguards web applications residing on a web server. The authors have claimed that this technique (AUSELSQI) can be used universally for any type of web server.

This technique examines the user input by intercepting request messages from HTTP. Examination contains looking for suspicious patterns or characters and then responding either by rejecting the request or passing it. The authors have implemented it as an ISAPI filter on IIS web server whereas it can also be implemented as firewalls or as component of existing web servers or as a standalone marshalling server.
C.4 SQLRand: Preventing SQL Injection Attacks

The authors have applied the concept of instruction set randomization to the SQL Language. The SQL keywords are changed by appending a random integer to them, which makes an attacker hard to predict. Therefore, attacker attempting an SQL injection attack would be discouraged. The design of SQLRand consists of a proxy which sits between the client and database server. The primary obligation of proxy is to de-randomize the random SQL query received and then conveys the SQL statements with the standard set of keywords to the back-end database for execution. If any malicious user attempts SQL injection attack, the proxy’s parser will fail to identify the randomized query and thus reject it.

To implement this approach, developer needs to modify client library. Authors have showed concrete evaluation report of performance overhead and proved that it doesn’t sacrifice performance.

C.5 JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications

Authors have proposed JDBC checker, static analysis tool to check the completeness of the dynamically built query strings. JDBC checker is used to identify programming errors in Java/JDBC applications. The tool is used to flag possible potential errors or identify the absence of errors in the dynamically built SQL queries. The analysis approach is the combination of two steps. First step is to generate finite state automaton which is then preprocessed to produce control flow graph labelled with the primitives, literals and keywords of SQL. Second step is to identify semantic errors such as SQL type error. This is achieved by applying Context free language reach-ability problem to test semantic checking. If semantic error is found, it is reported.
Authors have tested their tool in various code bases available from various sources like web, student team projects. Authors have argued that JDBC checker is successful in finding known and unknown errors in the program.

C.6 Securing Web Application Code by Static analysis and Runtime Protection

This technique detects input-validation related errors by performing information flow analysis. It secure web applications without user intervention by instrumenting vulnerable sections of code with runtime guards. Static analysis is performed to find out taint flows against preconditions for susceptible functions. The points in which such preconditions have not been fulfilled are detected and, sanitization functions and filters are suggested so that can be automatically appended to the application to persuade these preconditions. The primary drawback of this approach is that it assumes ample preconditions for susceptible functions can be accurately articulated using their typing system. The authors have implemented it in WebSSARI (Web application Security by Static analysis and Runtime Inspection) tool.

C.7 An Analysis Framework for Security in Web Applications

This technique is a static analysis framework which uses an abstract model of an application source program that conservatively estimate the set of SQL queries which a program may create as a finite state automation. Then it applies some checking algorithms on this finite state automaton to indicate or corroborate the absence of security infringements in the original web application program. It was not implemented in prototype when this paper was published. The authors were still working
on prototype building process.

C.8 Using Parse Tree Validation to Prevent SQL Injection Attacks

SQLGuard safeguards web applications by comparing parse tree structure of SQL query before inclusion of user input with the query resulting after the addition of user input.

It checks query at runtime to observe if it conforms to a model of expected query. The model is articulated as a grammar that only accepts legal queries and is generated at runtime by investigating the structure of the queries before and after the inclusion of user-input. A secret key is used to delimit user input by runtime checker during parsing, so security of this technique is fully dependent on the safety of secret key. The authors have implemented it in J2EE platform. In order to deploy this technique, developer has to either manually insert special indicators in the code where user input is inserted to a dynamically created query or rewrite code to employ a special intermediate library. Hence, an additional effort (modification of code base) is required to deploy it.

C.9 SQL Injection Protection by Variable Normalization of SQL Statement

Authors have proposed the variable normalization technique to protect web application against SQL injection attacks. First, the method use virtual database connectivity to extract basic structure of SQL statement. Secondly, this information is used to check the validity of SQL statement. If the SQL statement is different, it will be blocked. The method does not require modification of database application
source code.

The notion of this technique is to use variable normalization to modify the variables and preserve the structure of the SQL statement. Although user supplied variables (which basically differ every time) are used to construct SQL statement but the basic structure of SQL statement always remains same and if the supplied variables are malicious, it will change the structure of SQL statement and we will able to detect it.

The allowable list consists of the normalized SQL statement and variable requirement. The allowable list can be defined manually or it can auto learned as well. Proxy drive is used to normalize SQL statement and to check them against allowable list. If the normalized SQL statement exists, SQL statement will be allowed to execute otherwise it will be blocked. The authors have argued that performance will not be compromised with variable normalization.

C.10 Dynamic Taint Propagation for Java

Authors have proposed a dynamic solution for tagging and tracking user input at run time and preventing the improper use of untrusted user input (tainted data) to execute arbitrary commands. This technique can be applied directly to java classifier and it does not require source code modification. Authors have implemented their method in Java virtual machines and tested in number of real time application.

The authors have presented the method for tracking taintness of data originating from the untrusted user input, which is propagated throughout the life time of the application. To track the dynamic taintness of data, authors have specified following separation in the mechanism.

- Source: The user input originating from client is marked as tainted.
- Propagation: Strings derived from tainted data are also marked as tainted.
• Sink: A tainted string is prevented from reaching to the sinks.

The technique uses simple heuristic policy to untaint the tainted data. This depends on programmer to write meaningful input validation routines. The user input which passed input validation routines is not tainted. Otherwise, it raises exception and prevents a tainted string to reach to the sinks.

C.11 AMNESIA: Analysis and Monitoring for Neutralizing SQL Injection Attacks

The authors have presented model based approach for detecting and preventing SQL injection attack for Java based web applications. The authors have named their implementation, AMNESIA and they evaluate their technique on seven real time web applications.

AMENSIA uses the combination of static and dynamic analysis to counter against SQL injection attack. In static analysis part, it analyzes the application code and automatically generates the model of legitimate queries. In dynamic analysis part, it monitors all the dynamically generated queries at run time and compares them with the legitimate queries built with static model. If the query violates the model, it will be rejected from executing on the database. The technique consists of four steps.

• Identify hotspots: Scan the code to find hotspots points which issues SQL queries.

• Build SQL-query model: For each hotspot identified, build a SQL query model which represents all the possible SQL queries that may be built at that hotspot. It uses Java String Analysis to construct character-level automata, parse automata to group characters into SQL tokens.
• Instrument Application: At each hotspot, add function calls to the runtime monitor.

• Runtime monitoring: Check the dynamically generated queries against the static SQL query model and block queries that violate the model.

To implement this technique, developer doesn’t need to modify the source code of application. The authors have presented empirical results of accuracy and performance overhead.

C.12 Finding Security Vulnerabilities in Java Applications with Static Analysis

Authors have proposed static analysis tool based on scalable and precise context-sensitive pointer analysis for finding vulnerabilities such as SQL Injection, cross site scripting and HTTP splitting attacks caused by unchecked input. The users of the tool can specify vulnerability patterns using PQL (Program Query Language) which has java like syntax. The tool applies user specified queries to Java byte code and finds all possible matches statically. Our approach identifies all potential vulnerabilities matching the user provided specification in the statically analyzed code. The results of the analysis are then integrated into Eclipse (open source Java development environment), making the potential possible vulnerabilities easy to inspect and fix as part of the development process.

The main concept of the approach is to use information flow techniques to detect when tainted input has been used to construct an SQL query. These queries are then marked as SQL injection vulnerabilities. The analysis illustrates all the steps involved in passing taint from source to sink. This allows users to verify if any vulnerability found is exploitable. The authors have demonstrated the practicability of the tool using information flow techniques to detect security vulnerabilities in a
benchmark suite.

Authors have showed the experiment results to demonstrate the effectiveness and practicality of the tool for finding potential security vulnerabilities.

C.13 Finding Application Errors and Security Flaws Using PQL: a Program Query Language

SECURIFLY is a runtime security system for web applications that uses PQL (Program Query Language). This technique detects SQL injection vulnerabilities as well as other vulnerabilities such as path traversal attacks and cross-site scripting. It works by observing data flows. It tracks taint information on a per string basis rather than per character basis. Whenever necessary, it also sanitizes query strings that are produced using tainted inputs before they do any harm to the system. However, this sanitization process doesn’t help if injection attack is carried out into numeric fields. The primary drawback of this technique is to identify all the tainted input sources in highly-modular applications because it is a difficult task. Additionally, it is also difficult to accurately propagate taint information. This technique can be integrated with web server so that new web application is instrumented automatically whenever it is added to server.

C.14 SQL DOM: Compile Time Checking of Dynamic SQL Statements

Authors have named their implementation, SQLDOM. The basic concept of this method is to encourage the developer to use a set of classes which is strongly typed to database schema. Instead of using strings manipulation to build dynamic SQL queries programmed by developer, use safe API to generate SQL statements will take
care of security. It escapes malicious characters to prevent SQL injection attacks. SQLDOM is developed using C and .NET framework. SQLDOM consists of two parts.

- **Abstract object model**: Construct an object model that could be used to build every possible legal SQL statement which would execute at runtime.

- **Sqldomgen**: It is an API generation tool. It follows three main steps. First step is to obtain the database schema. The second step is to iterate through a table and columns of database schema and generate a number of files containing a strongly-typed instance of the abstract object model. The third step is to compile the generated output into Dynamic Link Library (DLL) that contains classes which are strongly-typed to a database schema. These classes are referred to as SQL DOM (SQL Domain Object Model).

The authors have tested SQLDOM and presented concrete data to check the accuracy and performance.

**C.15 A Learning-Based Approach to the Detection of SQL Attacks**

This technique is an anomaly based system which learns the profiles of expected database access carried out by web applications using a lot of different models. If any anomalous behavior is seen in the system, then it is considered as attack. This technique uses a number of different models to describe the profiles of benign access to the database. During a training phase, these profiles are trained automatically by analyzing sample database accesses. Then anomalous queries, considered as attack, are identified by system during the detection phase.
C.16  Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities

This technique is based on static analysis which detects common taint style vulnerabilities, such as SQL injection and cross-site scripting, automatically. It follows data flow analysis as main taint analysis while a supplementary literal analysis as well as alias analysis is also conducted to improve the precision and correctness. The authors have implemented it in Pixy tool (an open source Java tool) which only detects cross-site scripting flaws. Though the approach is targeted to detect both SQL injection and cross-site scripting, the implementation of it is only targeted for cross-site scripting.

C.17  SecuBat: A Web Vulnerability Scanner

Authors have developed generic, modular and open source web vulnerability scanner which they named SecuBat. SecuBat automatically analyzes websites and finds exploitable SQL injection and cross site scripting vulnerabilities. Authors have tested SecuBat on large number of real world websites and identified exploitable flaws in those websites. SecuBat uses a black box approach to crawl and scan websites to find exploitable vulnerabilities. SecuBat does not rely on signatures of known bugs. SecuBat consist of three main components.

- Crawling component: Collect the list of target websites.
- Attack component: Launches the configured attacks on these target websites. It scans each page for web forms which is the main entry point for web application.
- Analysis component: Examines the results return by application by using keywords and attack specific response criterion to verify whether an attack
was successful.

SecuBat is implemented on Windows .NET application in C using Microsoft SQL Server 2000 which stores all crawled and attack data. SecuBat provides Application Programming Interface (API) which allows programmers to apply their own modules for introducing other desired attacks. Authors have presented concrete analysis result of their test on real web application.

C.18 Automatic Revised Tool for Anti-Malicious Injection

Authors have argued that improper input validation is the root cause for the existence of vulnerabilities like SQL injection and cross site scripting. Authors have proposed the tool which can produce a proper validation function depending on application framework and database server. The tools first identify the entry point such as HTML forms, cookie variables, and types of scripting language etc. According to such information, tool will automatically generate a proper input validation function. The developer or application owner should use this input validation function into server side program.

The tool consists of four modules; spider, analyzer, function producer and tester. Spider is used to find entry point, analyzer to grab important variables, function producer to generate safe validation function and tester to check the completeness of validation function. Authors haven’t demonstrated their test results.
C.19 Eliminating SQL Injection Attacks - A Transparent Defense Mechanism

Authors have proposed the automated technique, which combines static analysis with run-time validation to protect against SQL injection attacks. The modification of source code is not required to deploy this technique (Only simple web server patch is required) and can be easily integrated with current existing system. The technique uses java string analysis library to perform static analysis on the source code. In static phase, it builds the intended SQL query structure in the form of SQL graph in the, and in the run time, SQL graph is validated against all the external inputs to capture the malicious query, and prevent it from being sent to the database to execute. Authors have presented the concrete evaluation results against different performance metrics.

C.20 Defending Against Injection Attacks through Context-Sensitive String Evaluation

Authors have presented an intrusion detection and prevention technique which they named as Context-Sensitive String Evaluation (CSSE). CSSE works by the combination of adding metadata (marking all user supplied data with metadata about its source), metadata preserving string operation (to make sure metadata are preserved and updated) and context sensitive string evaluation methods. CSSE does not require either source code modifications or developer interaction. Authors have implemented CSSE on PHP platform. Prior to execution, CSSE use context sensitive analysis to detect and reject SQL expressions to check if any SQL tokens has been built by malicious input. The basic idea is to modify the language design to differentiate user originated strings with static strings. Authors have evaluated
the CSSE with phpBB (open source bulletin board). They have demonstrated the concrete result of performance overhead.

**C.21 The Essence of Command Injection Attacks in Web Applications**

The authors have named their implementation, SQLCHECK. In the paper, authors have provided the first formal definition of command injection attack and SQL injection prevention technique. This prevention technique is based on context free grammars and comparing the parse tree constructed with user input at run time. They have evaluated SQLCHECK on real world application written in PHP and JSP. To evaluate SQLCHECK, list of real world attacks were used as input data.

SQLCHECK is built around the idea of blocking the queries in which user input can change the syntactic structure of query. To track the user input, authors have used meta-data to mark the beginning and end of each input strings. SQLCHECK is developed using grammar of output language and policy defining valid syntactic form of query. SQLCHECK resides on the web server. The web application then generates augmented queries through assignments, concatenations etc., which is parsed by SQLCHECK. If the query parses successfully, query is legitimate and SQLCHECK sends it without the meta-data to the backend database. Otherwise, SQLCHECK will block the query.

To implement this approach, developer either needs to rewrite the source code to use a special intermediate library or manually insert special meta-data markers into the code where user input is added to a dynamically generated query. The authors have provided concrete evaluation report of accuracy and performance overhead.
C.22 Preventing SQL Injection Attacks in Stored Procedures

This technique prevents SQL injection attacks in the database layer whereas most of the approaches prevent in application layer. It defends the attacks targeted against stored procedures. It works by combining static code analysis with runtime validation. In the static part, a stored procedure parser is designed and for any SQL query which relies on user inputs, it employs this parser to instrument the needed queries so that it can compare the original SQL query structure to that with user inputs. This technique can be deployed automatically and can be used only when needed.

C.23 Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications

This technique prevents SQL injection attacks in the database layer whereas most of the approaches prevent in application layer. It defends the attacks targeted against stored procedures. It works by combining static code analysis with runtime validation. In the static part, a stored procedure parser is designed and for any SQL query which relies on user inputs, it employs this parser to instrument the needed queries so that it can compare the original SQL query structure to that with user inputs. This technique can be deployed automatically and can be used only when needed.

C.24 Detecting Malicious SQL

This technique is a database level Intrusion Detection System (IDS). It is based on anomaly detection approach that checks SQL statements to detect SQL injection
attack and analyzes transactions to discover more intricate data-centric attacks. The intrinsic features of database applications are used which describes the abstraction of database utilization using two levels of detail; SQL commands and database transactions. The database interface, which is used for both learning and detection phased, captures the data flow between the web application and the database server. This IDS can be located inside the database server or in the local network close to the database server.

C.25 Using Aspect Programming to Secure Web Applications

This technique is based on the Aspect Oriented Programming (AOP) models. A security aspect called AProSec is designed for detecting both SQL injection attack and cross-site scripting. A distinct separation of the web application code and the security code is allowed with this aspect. It intercepts and validates all the requests from user to web server as well as all the requests from web server to database server. Moreover, this aspect can be parametrized. There is no need to recompile the source code by administrator so that he/she can freely choose which validations to use to each web application. The authors have described two experimentations, one with J Boss AOP and another with Aspect J Language.

C.26 SMask: Preventing Injection Attacks in Web Applications by Approximating Automatic Data/Code Separation

SMask is a novel approximation to data/code separation for detecting code injection attacks such as SQL injection, cross-site scripting etc. It uses string masking to
C.27 Sania: Syntactic and Semantic Analysis for Automated Testing against SQL Injection

This technique detects SQL injection vulnerabilities in web based applications during the development and testing phases. It intercepts the SQL queries that pass from application to database, and elaborate attacks are generated automatically according to the semantics and syntax of the potentially vulnerable locations in the SQL queries.

To identify the vulnerable locations, it analyzes the queries issued in reaction to the HTTP requests and identifies the vulnerable locations in queries in which an intruder can insert malicious strings. Additionally, parse trees comparison of the programmer intended SQL query and the query resulting after user inputs is performed to assess the security of these locations. This technique consists of a core component, an HTTP proxy and an SQL proxy. The HTTP proxy intercepts HTTP requests and responses while the SQL proxy intercepts the SQL queries. The authors have implemented it in a tool called Sania.
C.28 Automated Protection of PHP Applications against SQL-injection Attacks

Authors have proposed a novel automatic technique by combining static and dynamic analysis and code reengineering to protect PHP based web applications from SQL injection attacks. Static analysis parses the SQL queries and PHP code, dynamic analysis builds syntactic models of legal SQL queries and code re-engineering to protect legacy applications by inserting model based guard in PHP code from the SQL injection attacks. Authors have tested their technique in phpBB (PHP Bulletin board) Web application and results were promising. The technique produces very low false positive and false negatives and efficient success rate.

C.29 Using Automated Fix Generation to Secure SQL Statements

Authors have proposed an automated technique which will remove SQL injection vulnerabilities from Java based application. It will convert plain text vulnerable SQL statements into prepared statements. The developer can replace vulnerable code with generated secure prepared statement. It uses a SQL parse tree analysis to determine the structure of the SQL query and input variables. Authors have tested the technique on the five Java based program and showed that technique removed all the SQL injection vulnerabilities from the programs.
C.30 A Method for Detecting Code Security Vulnerability based on Variables Tracking with Validated-tree

This technique is based on finding SQL injection vulnerabilities by analyzing the program source codes. The notion of validated trees is used to track variables suggested by database operations in program scripts. The database operations are considered as secure or not by identifying whether the variables are manipulated by outside inputs. The final result is presented in a report which includes the location of SQL injection vulnerabilities in program scripts and the description of the related variables. It can be implemented universally in any web application platforms. The authors have implemented it in a tool called Software Code Vulnerabilities of SQL Injection Detector (CVSID).

C.31 Evaluation of Anomaly Based Character Distribution Models in the Detection of SQL Injection Attacks

This technique is an anomaly based approach which uses the character distribution of some parts of HTTP requests to defend web application from SQL injection attacks. It detects previously unseen SQLIAs so it is claimed to be more effective than other SQL defense techniques. It works by parsing the HTTP request’s query section and generates profile for each file. It does not require access to the source code or any modification to existing software modules. Two character distribution models; FCD (Frequency Character Distribution) and SCC (Same Character Comparison) are evaluated in this paper. This approach doesn’t require any user intervention for
C.32 Automated Fix Generator for SQL Injection Attacks

Authors have proposed the tool that automatically detects and fixes SQL injection vulnerable queries into safe SQL queries. The tool provides the solution using prepared statements. The tool is only targeted to PHP/MYSQL based web application and it is called securePHP. It has three major steps, vulnerability detection, creation of prepared statements, and report generation. GUI of securePHP is used to look for vulnerabilities. It uses grammar based violation principle (by parsing SQL statement) to detect the vulnerabilities in the query. User can generate reports for files containing vulnerabilities. Authors have used securePHP to test phpBB (open source bulletin board system) and found effective. The efficacy of the tool is presented in the paper.

C.33 SAFELI: SQL Injection Scanner Using Symbolic Execution

Authors have presented SAFELI, a tool for detecting SQL injection vulnerabilities in the web application. SAFELI is a short form of 'Static Analysis Framework for discovering SQL Injection vulnerabilities'. SAFELI instruments the byte code of the Java based web application and employ symbolic execution to statically examine security vulnerabilities. At each hotspot which submits SQL query, a hybrid string equation is generated to find out the original values of web controls which might lead to violation of database security. The solution provided by the hybrid string solver is used to generate a test case. If it encounters malicious SQL keywords,
it throws exception and halts the process. It will escape the malicious characters such as substitute single quotes with double quotes so that it cannot act as control characters.

SAFELI framework has four main components; Java Symbolic Execution Engine (JavaSye), Library of Attack Patterns, Hybrid String Solve, Test Case Replayer. Authors haven’t mentioned any evaluation results.

C.34 WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation

This technique was first published in 2006 and its updated version was published in 2008. It protects web applications from SQL injection attacks by using concept of positive tainting which is different from the conventional one (negative tainting).

Firstly, it identifies trusted data sources and marks data as trusted coming from these sources. Then it tracks trusted data at runtime using dynamic tainting and allows only the trusted data to shape as the semantically related part of SQL queries such as SQL operators and keywords. This technique also carried out syntax-aware evaluation of a SQL query string before the query string is sent to the database for execution. The authors have implemented this technique in WASP (Web Application SQL-injection Preventer) tool. This technique, in most cases, needs minimal or no developer interference to deploy it.

This technique follows a specification based methodology to detect SQL injection vulnerabilities. It utilizes specifications that characterize the programmer intended syntactic structure of queries which are generated and executed by the application. It also monitors the web application for executing SQL queries that are in breach of the specification. It filters the traffic between application server and database server where each SQL query passes through the validation process in order to check the potential existence of SQLIAs. This approach is independent of any specific target system, DBMS, or application environment. The authors have implemented it in SQL-IDS (SQL Injection Detection System) tool which monitors Java based applications. There is no need of source code modification in existing applications to use it.

C.36 An Automatic Mechanism for Sanitizing Malicious Injection

Authors have proposed the automatic defense mechanism for sanitizing malicious injection on a security gateway. It is hybrid approach and it reduces false rate. The technique performs white list, black list and encoding filtering method. White list is used first to allow known good input data to be passed. After that, Black list is used to block known bad input data and at last, special case will be sanitized. Authors have demonstrated the positive result of accuracy of the technique. Authors have argued that hybrid analysis is more efficient than single rule analysis.
C.37 SDriver: Location-specific Signatures Prevent SQL Injection Attacks

This technique incorporates a database driver that is placed between application and its underlying RDBMS (Relational Database Management System). This driver identifies each SQL statement by using the location of query and a stripped-down adaptation of its contents. These characteristics are analyzed during a training phase and a model of the valid queries is generated. At runtime, all queries are checked for acquiescence with the trained model. Thus queries containing maliciously injected components are blocked. During training phase, all the signatures are stored in an auxiliary database table. This driver also operates as a connectivity driver between the web application and the database driver. The authors have developed a tool called SDriver that implements this approach. For implementation, the application must be customized in the place where the web application sets up a connection with driver.

C.38 MUSIC: Mutation-based SQL Injection Vulnerability Checking

This technique is a mutation based testing approach for SQL injection vulnerability testing. In this technique, nine mutation operators that insert SQL injection vulnerability in web application source code are proposed. These nine operators later result in syntactic changes in the application source code which is known as mutants. The mutants can be killed only with test cases containing SQLIAs. An adequate set of test cases is generated to reveal more and more SQL injection vulnerabilities. This fault-based testing approach follows weak mutation testing rather than strong mutation testing. The authors have implemented this approach in MUSIC (Mutat-
tion based SQL injection Vulnerabilities Checking) tool that automatically produces mutants for the web applications written in JSP (Java Server Page).

C.39 Design Considerations for a Honeypot for SQL Injection Attacks

Unlike other techniques, this approach doesn’t defend against SQL injection attacks rather it attracts and gets injected with them. The authors have presented an application layer honeypot which attracts SQLIAs with the ultimate goal of directing the attacker to access false information. Honeytokens are used for this purpose which later can be used for tracking the attacker. Though the goal of this approach is being attacked, it restricts the attackers from escalating their attacks to operating system or performing attacks on other systems. To seem more genuine, it imitates the manifestation of common defenses against SQLIAs.

C.40 Combinatorial Approach for Preventing SQL Injection Attacks

Authors have proposed the unique technique to prevent SQL injection attacks in web application. The technique is based on the combination of signature based approach and auditing method. Authors have used Hirschberg Algorithm (divide and conquers approach) in signature based approach to reduce the time and space complexity. In this technique, signature based approach use three modules to detect the vulnerabilities.

• Monitoring module: To decide whether to send query statement for execution or not.
• Specification: It includes the predefined keywords and sends it to analysis module for comparisons. It uses DBMS auditing method for database transaction.

• Analysis module: To compare statement using Hirschberg algorithm obtained from specification module. It finds hot spot of the application and uses Hirschberg algorithm to compare strings which it received from monitoring module.

Since the proposed technique is defined at application level, it does not require any modification. Authors have not demonstrated any accuracy and evaluation results in the paper.

C.41 A Weight-Based Symptom Correlation Approach to SQL Injection Attacks

This technique is a multiple source approach. This technique collects symptoms of injections against web applications at different layers (particularly the application level and the network level) and correlates them via an approach that uses several different anomaly detection models. It combines data from multiple sources and convey such information which is different in nature. It analyzes logs and correlates the anomaly data extracted from them with inferred anomaly information in real-time by applications which helps to identify SQL injection attacks to raise an alert in the system.
C.42 An Approach for SQL Injection Vulnerability Detection

Authors have proposed an approach to detect the SQL injection vulnerabilities. They are focused on identification of user input manipulation vulnerabilities. Authors have used combination of static analysis, dynamic analysis and automatic testing in the approach. The approach is automated by a prototype tool 'SQLInjectionGen'.

Authors have used AMNESIA SQL query model and string argument instrumentation to trace the flow of input values. JCrasher is used to generate test cases and to change the test input with attack input. The approach generates color call graph indicating vulnerable and secure methods which will assist programmer to detect vulnerable locations in the program. Authors have evaluated their approach by performing case studies in two small web applications and results are demonstrated.

C.43 Automatic Creation of SQL Injection and Cross-Site Scripting Attacks

Authors have proposed a technique and automated tool Ardilla to detect the SQL injection vulnerabilities and cross site scripting. Authors have implemented the tool for PHP. The tool automatically creates inputs which exploit the SQL injection and cross injection vulnerabilities in the PHP/MySQL web applications. The tool is based on sample input generation, taint propagation (symbolically marks taints through execution) and input mutation to produce concrete exploits. Authors have argued that this tool is the first to precisely track the second order injection attack. Ardilla is the white box testing tool. It works without modification of source code of existing applications. It performs dynamic taint analysis to detect vulnerabilities.
One unique feature of Adrilla is that it tracks the flow of tainted data through the use of database. Adrilla has four components: Input Generator, Executor/Taint Propagator, Attack Generator/Checker and Concrete and Symbolic Database.

The user of tool needs to specify the type of attack (SQLI, first order or second-order XSS), the initial database state and the PHP program to analyze. The outputs of tool are attack vectors. The authors have evaluated the tool on five web application and presented the accuracy and performance results of the tool.

C.44 SQLProb: A Proxy-based Architecture towards Preventing SQL Injection Attacks

Authors have proposed SQLprob, the SQL proxy-based blocker to block SQL injection attacks. SQLprob is black box approach and does not require any modification of the code base or database of the application. It is independent of any programming languages and can be easily deployed to existing enterprise. SQLprob consist of four important components.

- The Query Collector: It processes all SQL queries during the data collection phase.
- The User Input Extractor: It identifies user input data using pair wise alignment algorithm.
- The Parse Tree Generator: It builds parse trees of all the incoming queries.
- The User Input Validator: It examines the user input to detect malicious input using user input validation algorithms. If it found to be malicious, it will be discarded, otherwise it will be sent to the database to execute.

Authors have evaluated SQLprob using real SQL attacks and presented the overhead and detection performance results of the tool. It protects all types of the SQL
injection attacks and incurs low performance overhead.

C.45 Shielding against SQL Injection Attacks using ADMIRE Model

Authors have proposed comprehensive and stepwise threat risk model ADMIRE to detect and counter the SQL injection attacks by shielding the underline database. The ADMIRE model analyzes risk assessment. The model follows six steps to counter against SQL injection attacks; Analyze the security objectives, Divide the application, Mark the vulnerabilities, Identify the threats, Rank the threat, and Eliminate the threat.

To mark the vulnerabilities, STRIDE (Microsoft Threat Model) has been used. STRIDE stands for Spoofing Identity, Tampering with data, Repudiation, Information disclosure, Denial of service and Elevation of privileges. To rank the threat, DREAD (Microsoft Ranking Model) has been used. DREAD stands for Damage Potential, Reproducibility, Exploitability, Affected Users and Discoverability. Threat model ADMIRE will provide the developers and designers a better understanding of the application and help to find bugs. It is a proactive approach and a vital part of the design process.

C.46 A Hybrid Analysis Framework for Detecting Web Application Vulnerabilities

Authors have developed a tool named PHAN (PHP Hybrid Analyzer), which detects the vulnerabilities in the web applications. It blends the combination of static and dynamic analysis to spot vulnerabilities on the web application which provides better run time overhead. In static phase, tool checks the PHP byte code to find
the dangerous code statements. In dynamic phase, only those statements which are found dangerous are monitored. PHAN only works on PHP byte code level instead of PHP source code.

In the static phase, it generates context free graphs for each program function. By analyzing CFGs, it detects statements which might affect the arguments of the sensitive sinks. It marks those statements dangerous which are only monitored in dynamic phase. It uses taint flow analysis to detect vulnerabilities in the dynamic phase. Authors have presented their evaluation result which was tested over some open-source PHP applications such as Clean CMS 1.5, Google CMS 1.8.2, MyForum 1.3, Pizzis CMS 1.5.1, W2B phpGreetCards and WordPress.

C.47 Multi-Layered Defense against Web Application Attacks

Authors have proposed a novel approach of multi layer defenses to the application level to detect SQL injection and cross site scripting attacks. The system architecture of technique has three components at abstract level:

- Filter: It filters special tags from malicious input.
- Detection module: It identifies malicious input through positive security (contains the signature of different attacks), negative security (contains the signature of allowed URL or SQL queries, tags in the form of regular expression) and anomaly detection components (monitors different attribute values of log entries).
- Analyzer Validation module: It performs syntactic and semantic validation. It produces Control Flow Graph and Validation Flow Graph (VFG) from the application code for input validation. It also generates validation report for developers.
Authors have evaluated the technique and demonstrated the accuracy and performance overhead results in the paper.

C.48 **SBSQLID: Securing Web Applications with Service Based SQL Injection Detection**

Authors have proposed a service based approach to detect SQL injection vulnerabilities. The main task of web service is to generalize the syntactical structure of query and validation of user inputs. Independent web service is placed in the system to parse the SQL statement. The three important module of this technique are:

- **Input validation**: It checks for the restricted character through pattern matching.
- **Query analyzer**: It checks for syntactic and semantic structure of the query.
- **Error service**: It makes error messages generalized to conceal the valuable information from the attackers and send to the application.

Authors have not described any evaluation results in the paper.

C.49 **Intrusion Detection in Web Applications: Evolutionary Approach**

This technique employs evolutionary meta-heuristic Gene Expression Programming (GEP) for the detection of intruders in web application. It is not only specific to SQL injection detection rather it detects other web intrusions as well. This technique transforms anomaly detection problem into classification problem with the objective of classifying SQL queries between normal and malicious queries. For classification, a function which is found out with the help of GEP is used. GEP and
its most important part; Genotype and Phenotype are presented descriptively. One the advantage of this approach is to find genuine solution very quickly.

C.50 On Automated Prepared Statement Generation to Remove SQL Injection Vulnerabilities

Authors have presented prepared statement replacement algorithm (PSR-algorithm) and an automatic fix generation tool for eliminating SQL injection vulnerabilities in the Java based web application. It generally replaces the vulnerable code with prepared statement. Authors have implemented PSR algorithm in Java as an automate fix generator called PSR generator. PSR generator takes in SQLIV Java file and outputs a java file with secure prepared statement code removing all the vulnerabilities. All the vulnerable Java files are replaced with the secure converted files.

Authors have described the algorithm details as well as the logic of generated code in the paper. Authors have evaluated their algorithm and presented the evaluation results.

C.51 A Novel Injection Aware Approach for The Testing of Database Applications

IAAT (Injection Aware Application Testing) is basically for testing the database applications. It tests the database applications for exposing SQL injection vulnerabilities under real conditions without use of any artifacts. A generic validation rule library is created and then, is customized for a particular application. The creation of the custom rule library is done at runtime in the existence of a real database.
Attack test cases are primed based on the custom library. Then a mock attack using these self generated test cases is commenced under the real conditions with the presence of real database. The basic framework of this approach is developed in Java while generic validation rules are written in XML.

C.52 CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks

This technique was first published in 2007 and its updated version was published in 2010. This technique works by comparing dynamically infer programmer intended SQL query structure against the issued actual SQL query structure.

For every user input, this technique creates benign sample inputs known as candidate inputs and the program is executed over actual inputs and sample inputs (candidate inputs) simultaneously. Then a candidate SQL query is created along with the actual SQL query where the candidate SQL query is always benign and actual SQL query is possibly malevolent. The actual query is rejected if parse structures of both queries do not match. The authors have implemented this technique in a tool called CANDID which defends SQL injection attacks by retrofitting web applications written in Java. It can also be implemented by modifying JVM (Java Virtual Machine) without requiring retrofitting.

C.53 TAPS: Automatically Preparing Safe SQL Queries

Authors have developed the automatic tool which can replace unsafe SQL queries of the legacy web application to use safe prepare statements. The use of prepared
statement to build query will remove the threat of SQL injection vulnerabilities in
the web application.

Authors have implemented their approach in the tool and which is called TAPS, Tool for Automatically Preparing SQL queries. Taps will replace all the unsafe
queries with equivalent PREPARE statements which is generated by a web appli-
cation. Taps generates derivation tree which leads to a query. Derivation tree is
the structure of symbolic expression of the query. Developer training is needed to
implement TAPS. It uses derivation tree to transform the legacy code to use pre-
pared statement. Authors have tested the tool in many real world applications and
presented the evaluation results in the paper.

C.54 Preventing Injection Attacks with Syntax
Embeddings - A Host and Guest Language
Independent Approach

Authors have proposed a syntax embedding, language independent technique to pre-
vent injection vulnerabilities in the web application. The notion of the technique is
to combine an API, from security perspective (to make sure that injection is not pos-
sible by construction) with conceptual ease of string manipulation. Authors achieved
this by embedding the syntax (grammars) of guest languages (e.g. SQL) into the
syntax of host languages (e.g. Java). The combined syntax is used by API generator
to generate an API that will manage the escaping, serializations and composition of
guest code sentences. Authors have implemented the technique in a prototype called
String-Borg. Authors have not presented any accuracy and performance overhead
results of the techniques explicitly. To implement this technique, developer training
is required.
C.55 A Heuristic-based Approach for Detecting SQL-injection Vulnerabilities in Web Applications

Authors have developed Viper, an automatic penetration testing tool for testing web application. It relies upon pattern matching of error messages and valid output pages. Therefore, it depends on extensive knowledge base of heuristics to guide the generation of SQL queries. The tool consists of four phases.

- Information gathering: Gathers information about structure of web application, identify the hyper-links and input forms.
- Identification of input parameters: Identify the web application input parameters within HTML forms.
- Generating attacks: Input fields are injected with SQL strings which are guided by heuristics of knowledge base.
- Result Reporting: It produces log files where all the information such as successful attacks, vulnerable pages, parameters, HTML forms, header etc is recorded.

Authors have presented the empirical evaluation results to prove its efficiency.

C.56 An Approach to Detection of SQL Injection Attack Based on Dynamic Query Matching

Authors have proposed the effective detection method (DUD) to prevent SQL injection attack which is based on dynamic query matching. DUD is independent of the developer’s static code checking, valid string database, initialization of syntactic
rules etc. It is a dynamic approach which detects the SQL injection by comparing
the queries. The technique will convert each query into XML form before matching
process. If matching fails, it will block the queries from reaching the database.

C.57 Access Control Mechanism for Web Databases
by using Parameterized Cursor

This technique, based on the concept of parameterized cursor, detects and prevents
SQL injection attack automatically. With the help of parameterized cursor, it uses
the fine-grain access method. It preserves a user session and stores it in database
with some other obligatory information like user-name password, user id, and date
of login. Whenever user tries to access the database, the stored user session is feed as
a parameter to the cursor. The query is executed only If the user is authenticated by
the cursor process else report the occurrence of SQL Injection attack. The security
of this technique is sturdy enough to block the execution of any SQL queries by
unauthorized user. This technique can be easily implemented and adopted in any
platform and database.

C.58 Use of Query Tokenization to Detect and
Prevent SQL Injection Attacks

The authors believed that user supplied data can change the intended structure of
query. Based on this idea, authors proposed query tokenization method to protect
SQL injection attack. The main principle of this method is the tokenizing of the
queries. The tokenization is performed on both the queries i.e. original query
and queries build with user input. The aim of tokenization process is to generate
tokens. These tokens are used to make an array. The length of the both array is
compared to verify the SQL injection attack. If the length of array is different, it will block the execution of the query. The authors have named their method Query Parser. Query parser should be executed first before the query will be used in the application. Authors have implemented approach in the Java platform. Authors haven’t mentioned any performance and accuracy results in the paper.

C.59 Artificial Neural Network based Web Application Firewall for SQL Injection

This technique is an application layer firewall based on Artificial Neural Network (ANN) that protects web applications against SQL injection attacks. This new approach is based on the ability of the ANN concept to carry out pattern recognition when it is suitably trained. A set of malicious and normal data is used to teach the ANN with Matlab during the training phase. The trained ANN is then integrated into a Web Application Firewall to protect the application during the operational phase. Two ANN filtering approaches: keyword based and characters based are used in this technique. A prototypic application firewall called ANNWAF has also been developed. It lies between the client and web application. The training engine can also be integrated into ANNWAF itself so there is no need of an external training engine.

C.60 D-WAV: A Web Application Vulnerabilities Detection Tool using Characteristics of Web Forms

This technique is an automated dynamic testing approach which detects web application vulnerabilities such as SQL injection attack, cross-site scripting, etc. It gets
a target web form with the help of given URL. It assigns test values to each field of web form after analyzing characteristics it. Then a method to create test suites which takes the credence of each test assessment into account is proposed. Finally, these test suites are executed and corresponding outcome based on response HTTP code and response HTML are analyzed. A Web Application Vulnerabilities Detection Knowledge Repository is used to find out whether the vulnerabilities exist or not. The authors have implemented this approach into D-WAV tool.

C.61 X-LOG Authentication Technique to Prevent SQL Injection Attacks

Authors have proposed the X-log authentication technique to prevent SQL injection attacks in the web based application. The main idea of this technique is to deploy three filtration techniques to prevent from SQL injections.

- Vulnerability Guard: To detect Meta characters, wild-card characters to prevent SQL injection attacks.
- X Log Authentication: To check the user input from X-Log Generator where a valid database is stored separately then the validated user input field is allowed to proceed.
- Stored Procedure: To check the size and data type of user input and to perform server side validation.

Authors have tested the technique on many web applications and argue that it prevents all kinds of SQL injection attacks. Authors have demonstrated the execution time of the queries and accuracy of the technique.