Detecting and Preventing SQL Injection Attacks: A Formal Approach

Mohammad Qbea'h, Mohammad Alshraideh, Khair Eddin Sabri
Computer Science Department
The University of Jordan
Amman, Jordan
Email: m.qbeah@ju.edu.jo, mshridah@ju.edu.jo, k.sabri@ju.edu.jo

***Abstract*—There are many organizations using databases to store and hide confidential data. Some of these data are published through World Wide Web (WWW) and the remaining data are hidden. Unfortunately, the attackers usually try to access and steal these hidden data by attacking the structure and the content of the database using an attacking technique called Structural Query Language Injection Attack (SQLIA). This technique gives the attackers illegal authorization to execute queries on database through the vulnerabilities in input boxes and page URL's. These queries may reveal or change the confidential data. Many techniques are available in the literature to prevent and detect SQLIA. However, these techniques do not consider languages other than the English language such as Arabic, Greek, and Japanese. Therefore, these techniques may not be able to discover attacks using such languages.**

In this paper, we present a formal approach to detect and prevent common types of SQLIA considering multi-languages. We formalize tautology and alternative encoding attacks using regular expressions and finite automata. We consider cases not discussed in the literature. Furthermore, we provide regular expressions and code in ASP.net which can be used by developers to detect and prevent attacks on websites that use Microsoft SQL server 2014 (MSSQL). We validate our work manually and by using tools. Results show that our model can detect and prevent SQL injection attacks including languages other than the English language.

***Keywords*-SQL Injection; Regular Expression; Finite Automata; Website Security ; Encoding**

I. Introduction

In these days, most of the data stored in servers are published in the internet through websites and web applications. These websites should reveal some data from database to the authorized users and hide the others. For authorization, users insert their user names and passwords through input boxes in the login pages which are sent to the database server. However, attackers may use these input boxes to sent SQL queries to the server to reveal hidden data or change the content of the database and its structure. This attack is called SQL injection attack (SQLIA).

SQLIA is a dangerious attack as significant portion of the vulnerabilities are in web applications [1]. Furthermore, Open Web Application Security Project (OWASP) made a study about web application security risks and it classified the top ten application security risks in 2013. The first one in the list is the injection risk such as SQL injection.

Most of the existing techniques that are used to detect SQLIA in websites such as [2], [3], [4], [5] did not discuss the attacks using languages other than the English language. Therefore, these techniques may not be able to identify attacks based on languages as Arabic, Greek, and Japanese. Furthermore, many models [3], [5], [6] did not take into consideration the alternate encoding type of SQLIA such as the unicode. This type of attack can reshape the attacking query into new syntax that could be considered by many models safe.

In this paper, we focus on the tautology and alternate encoding attacks. We characterize these attacks using regular expressions and finite automata. To present the applicability of our theory and to help developers preventing SQLIA, we develop regular expressions to detect injection attacks in ASP.net. The main contributions of this paper are:

- Considering languages in SQLIA other than the English language.
- Extending the tautology attacks by taking into consideration new operators such as "like" and <>.
- Discussing in details the prevention of alternate encoding attacks.
- Providing an ASP.net code that can be used by developers to detect and prevent the tautology and alternate encoding attacks.

The paper is organized as follows. In Section II, we summarize related work. In Section III, we identify some of the SQLIA and show their characterization as regular expressions and finite automata. In Section IV, we show the practical aspect of our proposed technique. Finally, we conclude in Section V.

II. Related Work

In this section, we summarize some of the existing works related to detecting or preventing SQLIA techniques. In [6], the authors proposed an approach using a static analysis combined with automated reasoning.

978-1-5090-2657-9/16 $31.00 © 2016 IEEE
DOI 10.1109/CCC.2016.26

This technique is effective for tautology attack only and cannot detect other types of SQL injection attacks. In [2], the authors proposed a technique to prevent SQL injection that uses static and dynamic phases based on pattern matching algorithm. Their technique uses user generated SQL query that is sent to the proposed static pattern matching algorithm. The result is efficient but the technique dose not consider all SQL injection attacks. In [3], the authors proposed a method to prevent SQL injection that divides the input query into tokens according to single quote, space, and double dashes. Then, it placed the tokens into dynamic tables for client and server. Both dynamic tables are compared and if both are equal, then there is no injection. Otherwise the query is rejected and there is an attack. However, this method did not take into consideration the ascii code, unicode, or hexadecimal encoding into consideration. The paper [4] proposed an approach for dynamic detection and prevention of SQLIAs. This approach which is based on regular expressions focuses on "trusted" strings. If the query is passed, then it is considered safe. Otherwise an attack is detected. However, their model do not consider the "like" operator in the tautology attack. Also, the model do not consider characters other than the English language. In [5], the authors used randomization to encrypt SQL keywords for detecting SQLIA. But their approach requires an additional proxy, computational overhead and remembering the keywords.

In [7], the authors proposed an authentication scheme that uses both Advanced Encryption Standard (AES) and Rivest-Shamir-Adleman (RSA) to prevent SQL injection attacks. In this method a unique secret key is assigned for every user. It uses a private key and public key for RSA encryption on side server. However, this technique requires to maintain keys and is not applicable to URL based SQL injection attacks.

The authors in [8] developed an approach that uses automatic test case generation to detect SQL injection vulnerabilities. The main idea behind this framework is based on creating a specific model that deals with SQL queries automatically. It also captures the dependencies between various components of the query. The used CREST (Automatic Test Generation Tool for C) test generator identified the conditions in which the queries are vulnerable. The authors in [9] suggested a technique to find vulnerabilities in web application such as SQL injection attack and Cross site scripting (XSS). They implemented this technique as an automated tool called Ardilla. This method uses static code analysis to find vulnerabilities. Also, it is applied on the source code of applications. It creates concrete inputs that expose vulnerabilities before an application is deployed.

A. Tools to detect SQLIA

There are many free tools to discover SQL injection. These tools are used to test and discover SQLI in websites but not used for prevention. We list some of the common tools. SQLMAP [10] is an open source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws. It comes with a powerful detection engine and many nice features. HAVIJ [11] is an automated SQL injection tool that helps testers to find SQL injection vulnerabilities on a web page. Users can use this tool on vulnerable web pages to perform back-end database fingerprinting, retrieve DBMS login names and password hashes, dump tables and columns, fetch data from the database, execute SQL statements against the server, and even access the underlying file system and execute operating system shell commands. SQL Inject Me [12] is a free tool that can be installed from Firefox adds-on, and can be used to search for SQL injection in the websites.

Burp Proxy [13] is an intercepting proxy server for security testing of web applications. It operates as a man-in-the-middle between user browser and the target application. It intercepts and modifies all HTTP/S traffic passing in both directions. This tool can be used with SQLMAP tool as a proxy to give important information about http protocol, user inputs, cookies, session id and other things. Backtrack5 [14] is a Linux-based tool that aids security professionals to perform assessments in an environment dedicated to hacking. By analyzing the mentioned tool, we found that these tools are not used for prevention and some of them may give incorrect results.

III. The Proposed Technique

In this section, we present the tautology and alternative encoding attacks. We give a comprehensive list of cases related to these attacks. Then, we formalize them as finite automata and regular expressions.

A. SQL Injection Attacks

There are two types of SQLIA detection. The first is static approach which is also known as pre-generating approach for detecting SQLIA. In this appraich, developers and programmers follow some guidelines for SQLIA detection during web application development. They use an effective validation and checking techniques for the input variable data. The other is dynamic approach which is also known as post-generated approach for detecting SQLIA. Post-generated technique are useful for the analysis of dynamic or runtime SQL query generated from user input data through a web application. This technique can detect SQLIA before a query is passed to the back-end or database server [3]. Our proposed technique is in line with this approach. We give expressions

that can be inserted into the code to validate user input and then detect and prevent the SQL injection attack.

1) Tautology: This attack allows accessing data through vulnerable input field or page URL by injecting SQL statements into the conditional query that are evaluated true. In this type of attack, many models focus on equality "=" operator only. However, there are other relational operators that give true results such as $(>, <, \geq, \leq, <>)$. For examples, the relations 3>1, 1≤ 5, 2<>1, 'A'<>'B' are true and therefore, confirmed tautology. In addition to these relational operators, the "like" operator can be used in this kind of attack as well. The tautology attack can be applied to input boxes or to page URL as illustrated below.

Table I: An example of a tautology attack in input box

Example	' 'ب' = 'ب' −− it is always true
Place	Input box
Normal query	Select username, password from users where username = 'Mohammad' and password ='P@ssw0rd'
Illegal reshaped query	Select username, password from users where username = 'Mohammad' or 'ب' = 'ب' −− and password ='P@ssw0rd'
Description	The attacker uses tautology command in user input box by using Arabic character 'ب' to login the system.

Table II: An example of a tautology attack in URL

Example	http://www.site.org/index.aspx?id=888' or '공격' = '공격' −− it is always true
Place	URL
Normal query	Select ID, password from users where ID = 888
Illegal reshaped query	Select ID, password from users where ID = 888 or '공격' = '공격' −−
Description	The attacker uses tautology command at the end of the URL by using Korean word '공격' which means "attack" in English to list all hidden ID's

2) Alternate Encodings: This type of attack is hidden and dangerous as it can reshape the syntax of the attack into a normal one that intruder detection systems may not detect. The SQL server 2014 uses alternate encoding with six operations: char, ascii, unicode, nchar, convert, and cast. The general syntax is presented in Table III.

Table III: Syntax of alternate encoding attacks

Operator	**General Syntax**
char	SELECT char(Number)
ascii	SELECT ascii('character')
unicode	SELECT unicode('character')
nchar	SELECT nchar(Number)
convert	SELECT convert(data type expression)
cast	SELECT cast(expression as data type)

The effectiveness of using alternate encodings in SQLIA is that the attacker reshapes the original SQLIA command to become a new normal syntax that contains embedded SQLIA. Then, the attacker executes the new normal syntax with additional statements that lead to abnormal results which can generate SQLIA. For example we succeed to write a query that contains implicit SQLIA as: select char(39) + char(32) + char(111) + char(114) + char(32) + char(49) + char(61) + char(49) + char(45) + char(45) as SQLIA. This query is equivalent to the tautology attack ' or $1 = 1 - -$.

B. Characterizing SQLIA

We formalize the presented attacks using finite automata and regular expressions. Each one has its applications. Our goal of this section is presenting the theory without considering its applications. We will consider one application in the next section. It is of note that both finite automata and regular expressions describe the same class of languages called regular languages. There are algorithms in the literature that can be used to convert each one of them to the other [15]. Figures 1 and 2 show the representation of the equality and the "like" tautology attack in FA. Attacks based on other operators (e.g., <, >) can be formalized similarly. The finite automata states that the attack can begin with any character but it should be followed by a comma and "or". Then, the attack should contain an equality between two strings. Finally, it should be ended with "−−". To cover characters that belong to different languages such as English, Arabic, Japanese, and others, we define the set of characters to include all characters such that $|\Sigma| = 2^{16}$ symbols. For readability and explanation, we ignored the case sensitivity for keywords and operators, but it is essential in practical.

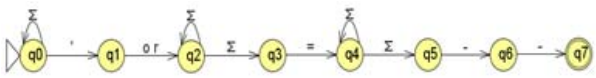


Figure 1: Formalizing tautology attack based on equality as FA

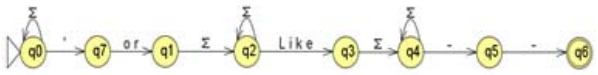
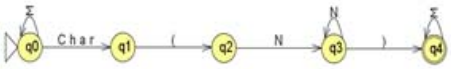


Figure 2: Formalizing tautology attack based on "like" as FA

The same attack can be formalized in regular expression as $(\Sigma^*$ ' or $\Sigma^+(= + <> + < + > + \leq + \geq + like)\Sigma^+) - -$. The operator Σ^* indicates any string of length 0 or more. The operator Σ^+ indicates any string of length 1 or more. The + operator is a choice which indicates in the above expression that the attack can be based on the operator =, <> or < etc.

We also formalize alternate encoding attacks which are mainly based on the operators char, asci, unicode, and cast in FA and RE. We show the formalism of the "char" operator. The other operators would be similar but different only in the data type of the operator. Table IV shows that the format of the attack usually contains the "char" operator with a parameter of type number.

Table IV: Formalizing alternate encoding attack based on char operator as FA and RE

FA	
' RE	$\Sigma^* char(N^+)\Sigma^*$

IV. Implementation and Experiment

We performed our experiment on Microsoft SQL server 2014 (MS-SQL) as a database back-end environment. Also, it was performed on Microsoft Visual Studio 2013 using ASP Dot Net with Visual Basic Dot Net programming language which are used for building websites and real applications. We used virtual studio environment to simulate, demonstrate, and test the SQLIA on local website. We focus on attacks against login web-pages through the input boxes and page URL's. Then, we used our approach to implement and generate practical regular expressions from the theoretical ones presented in previous section. Finally, we applied those expressions to the vulnerable code to detect and prevent attacks. It should be noted that our theory can be applied to other databases such as Oracle. The only difference is in the syntax of queries.

A. Preventing SQLIA

For our experiment, we build a vulnerable website. and then we insert to the code a regular expression in ASP.net based on the expressions given in the previous section. The site that we build contains two input boxes: one for user name and the other for password. We use the following vulnerable code in our testing:

```
conn.Open()
Dim cmd As New SqlCommand("select *
 from users where username ='" &
 TextBox1.Text & "' and password ='"
 & TextBox2.Text & "'", conn)
Dim dr As SqlDataReader = Nothing
dr = cmd.ExecuteReader()
 If dr.HasRows() Then
       Response.Redirect("~/default.aspx")
 End If
dr.Close()
conn.Close()
```

Before we give the expressions used to prevent attacks, we describe the syntax of the ASP.net regular expressions. The meaning of the RE operators written in ASP.net is given in Table V. A complete reference can be found in [16].

Table V: The meaning of some of the RE operators written in ASP.net

Symbols	**Description**
.	Match any character except newline
\w	Match any alphanumeric character
\s	Match any whitespace character
\d	Match any digit
\b	Match the beginning or end of a word
^	Match the beginning of the string
$	Match the end of the string
*	Repeat any number of times, 0 or more occurrences.
+	Repeat one or more times
?	Repeat zero or one time.
|	or

We can detect the tautology attack using the following regular expression

```
"([\w]*|[\W]*)(')[1][\s]*(or)[1](\s)[1][\s]*
(')[0,1]([\w]*|[\W]*)+(')[0,1][\s]*(=|like|
<|>|>=|<=|<>)[1][\s]*(')[0,1]([\w]*|[\W]*)+
(')[0,1][\s]*[;]*[\s]*[-][2][\s]*$"
```

The symbol \w matches any word character while \W matches any non-word character. The combination of \w and \W gives all alphabets = 2^{16} unicode characters. This allows us to detect attacks in languages other than the English language. This regular expression corresponds to the one given in the previous section as shown in Table VI.

Table VI: The correspondence between theoretical RE and RE in ASP.net

Theoretical RE	RE is ASP.net
Σ^*	`([\w]*|[\W]*)`
or	`(or){1}`
−−	`[-]{2}`
+	|

The expression presented previously can be used to detect tautology attacks. Also, the expression can be used to prevent the attack by adding the following code segment

```
Dim TautRegex As String = "([\w]*|[\W]*)(')
{1}[\s]*(or){1}(\s){1}[\s] *(')
{0,1}([\w]*|[\W]*)+(')
{0,1}[\s]*(=|like|<|>|>=|<=|<>){1}[\s]*(')
{0,1}([\w]*|[\W]*)+(')
{0,1}[\s]*[;]*[\s]*[-]{2}[\s]*$"
If Regex.IsMatch(TextBox1.Text, TautRegex) Then
  ResultLabel.Text = "Invalid input, you
    are trying to use SQL Injection Attacks...!"
  Exit Sub
Else
  ResultLabel.Text = "Valid input."

End If
```

We use our theory to prevent alternate encoding attack as well. The SQL server 2014 uses alternate encoding with six operations: char, ascii, unicode, nchar, convert, and cast. This attack can be prevented by adding the following code.

```
Dim AsciiRegex As String = "([\w]*|[\W]*)(ascii)
 {1}[\s]*[(]+[\s]*(')
 {0,1}([\w]*|[\W]*)+(')
 {0,1}[)]+[\s]*([\w]*|[\W]*)"
Dim UnicodeRegex As String = "([\w]*|[\W]*)
 (unicode){1}[\s]*[(]+[\s]*(')
 {0,1}([\w]*|[\W]*)
 +(')
 {0,1}[)]+[\s]*([\w]*|[\W]*)"
Dim CharRegex As String = "([\w]*|[\W]*)(char){1}
 [\s]*[(]+[\s]*(')
 {0,1}([\w]*|[\W]*)+(')
 {0,1}[)]
 +[\s]*([\w]*|[\W]*)"
Dim CastRegex As String = "([\w]*|[\W]*)(cast){1}
 [\s]*[(]+[\s]*(')
 {0,1}([\w]*|[\W]*)+(')
 {0,1}[)]
 +[\s]*([\w]*|[\W]*)"

If Regex.IsMatch(TextBox1.Text, AsciiRegex)
      OrElse
 Regex.IsMatch(TextBox1.Text, UnicodeRegex)
      OrElse
 Regex.IsMatch(TextBox1.Text, CharRegex) OrElse
 Regex.IsMatch(TextBox1.Text, CastRegex) Then
    ResultLabel.Text = "Invalid input, you are
      trying to use SQL Injection Attacks...!"
Exit Sub
Else ResultLabel.Text = "Valid input."
End If
```

Another way of attacks through encoding is the use of hexadecimal. For example, the hexadecimal of equality "=" is "%3D". This technique is used in URLs. We can use the hexadecimal encoding to produce attacks such as tautology. The common syntax of tautology attack ' or 1=1−− can be represented in hexadecimal as %27 %20 %6F %72 %20 %31 %3D %31 %2D %2D where for example %20 represents space and %6F represents the character o.

```
"([\w]*|[\W]*)((')
{1}|(%27){1})([\s]*|(%20)*)
((or){1}|(%6F%72){1}|(o%72){1}|(%6Fr){1})
([\s]+|(%20)+)((')
{0,1}|(%27){0,1})
([\w]*|[\W]*)+((')
{0,1}|(%27){0,1})
([\s]*|(%20)*)((=){1}|(%3D){1})([\s]*|(%20)*)
((')
{0,1}|(%27){0,1})([\w]*|[\W]*)+
((')
{0,1}|(%27){0,1})([\s]*|(%20)*)([;]*|(%3B)*)
([\s]*|(%20)*)([-]|(%2D)){2}([\s]*|(%20)*)$"
```

B. Testing

In this section, we provide two ways to validate and test the effectiveness of our code.

1) Manual Testing: we insert SQLIA to the login input boxes before and after inserting our prevention code. The test shows that our code prevents tautology attacks even considering languages other than the English language as shown in Figures 3 and 4. This is because we use \w and \W to consider all unicode characters. Other models such as [4] consider only the range of English characters and numbers.

Figure 3: SQLI using "=" with Arabic character

We also apply alternative attack using the cast operator. For example, by using the cast operator, the following code

```
DECLARE @i VARCHAR(8000);SET @i = CAST(
0x555345205B6D61737465725D20414C5445522044415
44142415345205B746573745D20534554202052454144
```

Figure 4: SQLI using "like" with numbers

```
5F4F4E4C592057495448204E4F5F574149543B
AS VARCHAR(8000));EXEC(@i);
```

has the same effect as

```
USE [master] ALTER DATABASE [test] SET
READ_ONLY WITH NO_WAIT
```

which changes "test" database to read only. This code can be obtained by using the following command

```
SELECT CAST('USE [master] ALTER DATABASE [test]
   SET  READ_ONLY WITH NO_WAI;'
   AS VARBINARY(8000));
```

We insert the command into the input box before adding the preventing code. It changes the database to read only as shown in Figures 5 and 6. However, after adding the regular expression code, we can detect and prevent this attack as shown in Figure 7

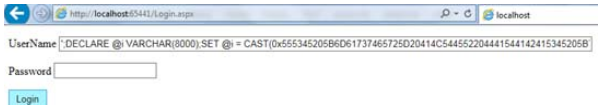


Figure 5: An attack based on the cast operator

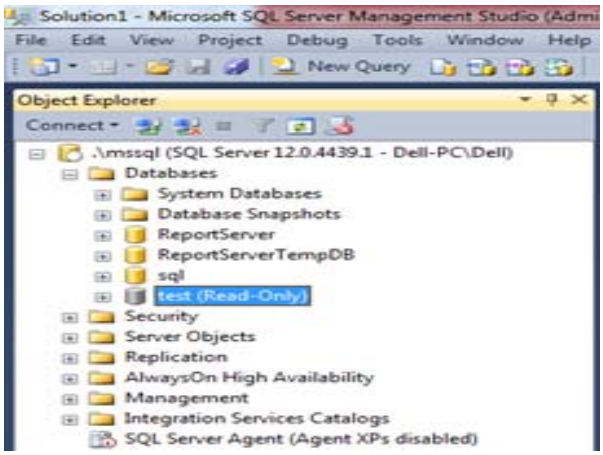


Figure 6: The result of the attack

Finally, we test our code against the hexadecimal encoding. We show in Figure 8 that inserting the code

Figure 7: Preventing the cast operator attack

%27 %20 %6F %72 %20 %31 %3D %31 %2D %2D, which has the effect of ' or 1=1−−, can reveal confidential data. Figure 9 shows that our regular expression can prevent this attack.

Figure 8: An attack based on hexadecimal encoding

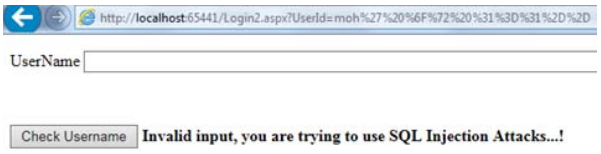


Figure 9: Detecting Hexadecimal Attack

Regarding the performance of detecting the attacks, we have inserted all the prevention expressions into our vulnerable code. Then, it was experimented using several inputs. The experiment shows that the time required to validate the input is not noticeable. This is because the finite automata corresponding to the regular expression does not have many branches. Furthermore, the length of the finite automata is not long.

2) Tool: The "Sculptor" tool [17] is used to validate the effectiveness of our code. We select Sculptor rather than other tools as it allows us to test our web page against the tautology and alternate encoding attacks. "Havij" tool does not test web pages against tautology or alternate encoding attacks. It can be used to test pages against other attacks such as blind attack. The "SQLMAP" tool does not allow the selection of a specific kind of attack.

We use the Sculptor tool to test our web page against the tautology and alternate encoding attacks. First, we test the page before inserting the prevention code. In this case, the Sculptor tool shows that the web page is vulnerable to attacks. However, after we insert the code, the tool shows that the web page is not vulnerable to tautology or alternate encoding attacks. Using the

tool, in addition to the manual testing, ensures the effectiveness of our code to prevent attacks.

V. Conclusion

In this paper, a formal technique to detect and prevent SQLIA is presented. This technique can help researchers, developers, and programming languages designers to detect and prevent SQLIA. We compounded theoretical method based on FA and RE with the practical aspect by developing ASP.net code based on our expressions. The code which is used to prevent attacks is tested manually and by a tool. The results show that the code prevents attacks against web pages. This code can be used by developers to secure their web-sites without restructuring or rewriting their code.

Our expressions prevent attacks only. For example, all the strings "or", "'or", "'or 1", "'or 1=" are not considered as attacks and can be used in users' password. Therefore, our technique gives users more flexibility to choose their passwords by preventing real attacks expressions instead of prohibiting the use of "=" for instance. Our technique considers characters other than English characters. Also, it covers cases in tautology attack in addition to the "=" relational operator such as the "like" operator. Furthermore, the tool detects attacks based on alternate encoding such as cast operator.

Our theory can be applied on programming languages other than ASP.net such as JAVA and PHP. Also, it has applications other than the one presented in the paper as building intruder detection systems or generating SQL injection attacks. As a future work, we intend to characterize and formalize more SQL injection attacks.

References

[1] R. Johari and P. Sharma, "A survey on web application vulnerabilities (SQLIA, XSS) exploitation and security engine for sql injection," in *Proceedings of the 2012 International Conference on Communication Systems and Network Technologies*. IEEE Computer Society, 2012, pp. 53–458.

[2] M. A. Prabakar, M. KarthiKeyan, and K. Marimuthu, "An analysis framework for security in web applications." in *Proceedings of the International Conference on Emerging Trends in Computing, Communication and Nanotechnology (ICE-CCN)*. IEEE Computer Society, 2013, pp. 503 – 506.

[3] S. Anjugam and A. Murugan, "Efficient method for preventing sql injection attacks on web applications using encryption and tokenization," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 4, no. 4, pp. 173–177, 2014.

[4] A. Sravanthi, K. J. Devi, K. S. Reddy, A. Indira, and V. S. Kumar, "Detecting SQL injections from web applications," *International Journal of Engineering Science & Advanced Technology*, vol. 2, no. 3, p. 664 – 671, 2012.

[5] S. W. Boyd and A. D. Keromytis, "SQLrand: Preventing SQL injection attacks," in *Proceedings of the Applied Cryptography and Network Security Conference*, ser. Lecture Note in Computer Sciences, vol. 3089, 2004, pp. 292–302.

[6] G. Wassermann and Z. Su, "An analysis framework for security in web applications." in *Proceedings of the Workshop on Specification and Verification of Component-Based Systems SAVCBS '04*, 2004, p. 9.

[7] I. Balasundaram and E. Ramaraj, "An authentication scheme for preventing SQL injection attack using hybrid encryption (PSQLIA-HBE)," *European Journal of Scientific Research*, vol. 53, no. 3, pp. 359–368, 2011.

[8] M. Ruse, T. Sarkar, and S. Basu, "Analysis & detection of SQL injection vulnerabilities via automatic test case generation of programs," in *Proceedings of the 10th IEEE International Symposium on Applications and the Internet (SAINT)*, 2010, pp. 31 – 37.

[9] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of SQL injection and cross-site scripting attacks," in *Proceedings of the 31st International Conference on Software Engineering ICSE*. IEEE Computer Society, 2009, pp. 199–209.

[10] B. D. A. G. and M. Stampar, "sqlmap: automatic sql injection and database takeover tool," 2013, http://sqlmap.org/ (accessed on July, 2016).

[11] ITSecTeam, "Havij advanced sql injection," 2013, http://itsecteam.com/products/havij-advanced-sql-injection (accessed on July, 2016).

[12] "Sql inject-me," https://labs.securitycompass.com/exploit-me/sql-inject-me/ (accessed on July, 2016).

[13] Portswigger, "Burp proxy," http://portswigger.net/burp/proxy.htm3 (accessed on July, 2016).

[14] "Backtrack-linux," http://www.backtrack-linux.org/ (accessed on July, 2016).

[15] M. Sipser, *Introduction to the Theory of Computation*, 3rd ed. Cengage Learning, 2012.

[16] "MSDN," https://msdn.microsoft.com/en-us/library/az24scfc%28v=vs.110%29.aspx (accessed on July, 2016).

[17] "Sculptor," http://www.sculptordev.com (accessed on July, 2016).