On Preventing SQL Injection Attacks

Bharat Kumar Ahuja, Angshuman Jana, Ankit Swarnkar, and Raju Halder

Indian Institute of Technology Patna, India
{bharat.cs10, ajana.pcs13, ankitswarnkar.cs10, halder}@iitp.ac.in

Abstract. In this paper, we propose three new approaches to detect and prevent SQL Injection Attacks (SQLIA), as an alternative to the existing solutions. These techniques are (*i*) Query Rewriting-based approach, (*ii*) Encoding-based approach, and (*iii*) Assertion-based approach. We discus in details the benefits and shortcomings of the proposals *w.r.t* the literature.

Key words: SQL Injection Attacks, Query Rewriting, Encoding, Assertion Checking

1 Introduction

SQL injection is an attack in which malicious SQL code is inserted or appended into database application through user input parameters that are later passed to a back-end SQL server for parsing and execution [11]. The growing popularity and intense use of database-driven web applications in today's web-enabled society make them ideal target of SQL Injection Attacks (SQLIA). The number of SQLIA has increased rapidly in recent years, and it has become a predominant type of attacks. Consider the following PHP script from [4]:

```
\\ Connect to the database
        $conn = mysql_connect("localhost", "username", "password");

\\ Dynamically building sql statement with user input
        $query="SELECT * FROM Products WHERE Price<'$_GET["val"]'".
                                        "ORDER BY ProductDescription";
\\ Executing the query against the database
        $result = mysql_query($query);
```

The URL "`http://www.victim.com/products.php?val=100`" is used to view all products with cost less than $100. However, on providing malicious input **100' OR '1'= '1** and the corresponding URL `http://www.victim.com/products.php?val=` **'100' OR '1'='1'**, the dynamically created SQL statement "`SELECT * FROM Products WHERE Price < '100' OR '1'= '1' ORDER BY ProductDescription`" extracts all product information as the `WHERE` clause evaluates to true always.

In addition to the above tautology-based SQL injection attacks, there exist various other forms of attacks with various attacker intents. For instance, Union Query, Piggy-Backed Query, Stored Procedures, etc. [11].

The primary reason behind SQL injection is the direct insertion of code into parameters that are finally concatenated with SQL commands and executed. When Web applications fail to properly sanitize the inputs, it is possible for an attacker to alter the construction of back-end SQL statements, which may lead to a catastrophe. Web applications along with databases, therefore, require not only careful configuration and programming to ensure data security but also need an effective and efficient protection technique to prevent SQL injection attacks.

A wide range of prevention and detection techniques are proposed to address the SQL injection problems [11], [4]. According to the best of our knowledge, no existing approach can guarantee a complete safety. For example, many proposed approaches, like AMNESIA – a model-based technique [9], Taint-based technique [17], Intrusion Detection System [22], etc. have large number of false positive alarm. The Static Code Checker [7] is used to prevent tautology based attack only. The approach of Instruction Set Randomization [2] is unable to prevent many types of SQLIA like Illegal/Logically Incorrect Query, Stored Procedures, Alternate Encodings etc. It is extremely difficult to apply defensive coding practices [14] for all the sources of inputs because it results high rate of false positive (e.g. in case of input O'Brian).

In this paper, we propose three new directions to detect and prevent SQLIA, as an alternative solutions. These are (*i*) Query Rewriting-based approach, (*ii*) Encoding-based approach, and (*iii*) Assertion-based approach.

The main objective of first approach is to mitigate the effect of concatenation operation during query generation. The objective of the second approach is to mitigate the mal-effect of bad input on query semantics. The assertion-based technique is a state verification technique which can be performed either at application level or at database level. We provide a comparative study among our proposed approaches based on security guaranty, usability and applicative point of view.

The structure of the paper is as follows: Section 2 discusses related works in the literature. Section 3 describes and formalizes the problem. We introduce our proposed approaches in section 4. Section 5 represents the comparative study and complexity analysis. Section 6 concludes the paper.

2 Related Works

In [10], authors proposed a model-based technique to prevent SQLIA that combines both static and dynamic analysis. In static phase, the approach builds character-level Non-Deterministic Finite Automaton (NFA) model for each hotspot. All dynamically generated queries, during runtime, are checked against the corresponding model and violation of the models is reported as SQLIA. However, the success of this approach is dependent on the accuracy of its static model, and may results both false positives and false negatives. Other approaches on static models considering grammars and parse-trees are proposed in [21, 3].

Defensive Coding Practice [14] is a way to prevent the SQL injection vulnerabilities. But it is extremely difficult to apply for all sources of input, because in many applications SQL-keywords, operators can be used to express normal text entry, formulas or even names (e.g. O'Brian).

In [2], authors proposed Instruction Set Randomization approach and introduce the SQLrand tool. It provides a framework that allows developers to create queries using randomized instructions instead of normal SQL keywords. The attacker is not aware about that randomize instruction. Thus if any malicious user attempting to SQL injection attack would immediately be thwarted. However, it con not cover all types of SQLIA. It is unable to detect or prevent the many types of SQLIA like Illegal/Logically Incorrect Query, Stored Procedures, Alternate Encodings etc. and also it imposes a significant infrastructure overhead.

In [8], authors introduced an obfuscation-based approach to detect the presence of possible SQLIA. Obfuscation removes the need of concatenations operation and treats the atomic formulas in the condition part of the queries as independent from each other, whereas the dynamic phase, after merging the user inputs in the obfuscated atomic formulas, verifies the the presence of possible SQLIA.

CANDID [1] is a code transformation-based approach which aims to construct programmer-intended query structure. In order to construct the intended query, the approach runs an application on a set of candidate inputs that are self-evidently non-attacking. However, false positive is the prime limitation of this approach.

In [18], authors proposed an idea to identify various types of SQLIA and to mitigate such attacks by redefining code-injection attacks on outputs (CIAOs). Precisely, detecting CIAOs basically follows the dynamic white-box mechanisms which are based on dynamic taint analysis. The primary limitation is that determining whether an application is vulnerable to CIAOs requires the knowledge about which input symbols are propagating to the output program.

DIGLOSSIA [20] is a run time tool which performs a dual parsing to compare the shadow query (which must not be tainted by user inputs) with the actual application generated query, and ensures that the query issued by the application program does not contain injected code. However, this does not consider all sources of inputs (*e.g.* it does not permit any input to be used as a part of the code in the query).

Another two techniques SQL DOM [16] and Safe Query Object [6] are used for encapsulation of database queries to provide a safe and reliable way to access databases. The main limitation of this technique is that they require developers to learn and use a new programming paradigm or query-development process.

Several automated or semi-automated tools for detection and prevention of SQLIA are already developed. For instance, AMNESIA [9], SQLCheck [21], SQLGuaed [3], SQLrand [2], WebSSARI [12], JDBC-Checker [7], etc.

3 Problem Description

The characteristics of dynamic web applications are:

- They receive inputs as strings during run-time.
- They build SQL query strings by performing concatenation operation between SQL constructs and input strings.
- Bad input, after concatenation, may be treated as a part of SQL control constructs, leading to SQL Injection Attacks.

Formally, this can be defined as follows [21]: Let Σ be an alphabet. A web application P: $(\Sigma^* \times \Sigma^* \times \cdots \times \Sigma^*) \rightarrow \wp(\Sigma^*)$ is defined as a mapping from a set of user inputs over an alphabet Σ to a set of query strings of Σ. Given a set of SQL sub-strings $\{ s_1, s_2, \ldots s_n \}$ and a set of input strings $\{ i_1, i_2, \ldots i_m \}$, P generates a query string (using concatenation operation) $Q = q_1 + q_2 + \ldots + q_l$, where for $1 \leqslant j \leqslant l$:

$$q_j = \begin{cases} i & where \;\; i \in \{i_1, i_2, \ldots, i_m\} \\ s & where \;\; s \in \{s_1, s_2, \ldots, s_n\} \end{cases}$$

Given a query string Q, it can be divided into two parts: data-part and control-part. Let us denote $Q = \langle \{d_1, d_2, \ldots, d_p\}, \;\; \{c_1, c_2, \ldots, c_q\} \rangle$ where $\{d_1, d_2, \ldots, d_p\}$ is the set of data-parts and $\{c_1, c_2, \ldots, c_q\}$ is the set of control-parts. SQL Injection attack occurs iff $\{c_1, c_2, \ldots, c_p\} \cap \{i_1, i_2, \ldots, i_m\} \neq \phi$; that is, when any input string is treated as SQL control construct in Q.

4 Proposed Approaches

In this section, we propose three possible solutions as an alternative to the existing ones.

4.1 Query Rewriting Approach

In database-enabled web-applications, SQL Queries are, in general, constructed by concatenating input strings directly from the users. This helps attackers to somehow manage and modify the query structures by providing malicious SQL keywords in the input strings. In order to mitigate the use of concatenation operation, we propose a query-rewriting approach which transforms the insecure web application into semantically secure version. The proposed approach has following two main phases:

- Static Phase
  - Insert user inputs into a database table, called "`Input`" table.
  - Replace the concatenation operation of user inputs with other SQL constructs by an equivalent `SELECT` query which selects inputs from the table "`Input`".
- Dynamic phase
  - Parse the `INSERT` statement after merging inputs to check the correctness of syntax.

Static Phase:

In this phase, all the inputs of the web-application are inserted into a database table, called "`Input`" table, using `INSERT` statement. The objective is to mitigate the bad effect of input data by treating them as a part of the database and to remove the necessity of concatenation operation by using them during the query formation later on.

The generic structure of the "`Input`" table is as follows: Let the web application P involves n queries $\{ Q_i \mid i = 1, \ldots n \}$. Suppose each query Q_i accepts m user inputs $S_I^{Q_i} = \{ I_{ij} \mid j = 1, \ldots m \}$. The structure of "`Input`" table is

`INPUT(QID` *integer*, `IID` *integer*, `u_input` *varchar*)

where `QID`, `IID` represent program point i and input id j, which together uniquely identify j-th input I_{ij} in i-th query Q_i – hence forms primary key.

Static Analyzer. Let l_i: Q_i ($\{I_{ij} \mid j = 1, \ldots, m\}$) denotes that query Q_i is formed using concatenation operation with user inputs $\{I_{ij} \mid j = 1, \ldots, m\}$ at program point l_i. The analyzer scans the web application and performs the following operations:

1. Identify l_i: Q_i ($\langle I_{ij} \mid j = 1, \ldots, m \rangle$) for i=1, $\ldots$ n.

2. Before each l_i, add an `INSERT` statement which inserts all inputs $\{ I_{ij} \mid j = 1, \ldots m \}$. For example, consider the following query:

   "`SELECT eid FROM emp WHERE login=' "` +slogin+ "`' AND pass=' "` +spass+ "`'"`

   The analyzer adds two `INSERT` statements before the query, assuming it is at program point 1 as follows:

   "`INSERT INTO INPUT(GID, IID, u_input) VALUES ('1', '1', ' "` +slogin+ "` ')"`

   "`INSERT INTO INPUT(GID, IID, u_input) VALUES ('1', '2', ' "` +spass+ "` ')"`

   "`SELECT eid FROM emp WHERE login=' "` +slogin+ "`' AND pass='"` +spass+ "`'"`

3. Convert the query Q_i by semantically equivalent version where each concatenation operation and input variables are replaced by corresponding `SELECT` query accessing the same values from the "`Input`" table.
   For example, the query mentioned just before is transformed as:

   "`INSERT INTO INPUT(GID, IID, u_input) VALUES ('1', '1', ' "` +slogin+ "` ')"`

   "`INSERT INTO INPUT(GID, IID, u_input) VALUES ('1', '2', ' "` +spass+ "` ')"`

   ```
   SELECT eid FROM emp WHERE login = (SELECT u_input FROM INPUT WHERE QID = 1 AND
          IID = 1) AND pass = (SELECT u_input FROM INPUT WHERE QID = 1 AND IID = 2)
   ```

Observe that, a basic filtering is applied on inputs to filter the presence of any meta-character which is treated as control-character of `INSERT` statement. For instance, ' is replace by ".

Dynamic phase:

In the modified version of web application, it is observed that only `INSERT` statement is vulnerable to SQLIA. Attacker may try to change the behaviour of `INSERT` statement through malicious inputs. A Dynamic Analyzer will check the correctness of the syntax of `INSERT` statement after merging inputs and before issuing to the database. For this purpose we define the following grammar for `INSERT` statement:

$$
\begin{aligned}
\texttt{Ins\_ stmt} &::= \text{INSERT INTO } \texttt{E S} \\
\texttt{E} &::= \texttt{Identifier} \\
\texttt{S} &::= \texttt{(attr X val)} \\
\texttt{X} &::= \texttt{, attr X val , |} \text{) VALUES (} \\
\texttt{attr} &::= \texttt{Identifier} \\
\texttt{val} &::= \texttt{' id '} \\
\texttt{id} &::= (\text{string} \mid \text{number} \mid \text{com} \mid \text{meta})^* \mid \text{null} \\
\texttt{Identifier} &::= \text{letter}(\text{letter} \mid \text{digit})^* \\
\texttt{com} &::= = \mid > \mid < \mid \geqslant \mid \leqslant \mid != \\
\texttt{meta} &::= \text{`} \mid \text{'} \mid - \mid ;
\end{aligned}
$$

Observe that, these production rules check the number of inputs with the number of attributes in the `INSERT` statement. We can also add a mechanism for type checking, in addition.

Illustration with Example.

Let us consider the database and web application depicted in Figure 1(a) and 1(b) respectively.

Static phase: As discussed, the static analyzer adds `INSERT` statement aiming at inserting all inputs into the "`Input`" table and replaces the input strings along with the concatenation operation by equivalent `SELECT` queries accessing data from "`Input`" table. The result is shown in Figure 1(c).

Dynamic phase: Consider the code in Figure 1(c) and the inputs: `user` and `x'OR 1=1`: After merging the inputs, the `INSERT` statement is

3a. `INSERT INTO INPUT (QID, IID, u_input) VALUES ('3', '1', ' user ' )`

3b. `INSERT INTO INPUT (QID, IID, u_input) VALUES ('3', '2', ' x''OR 1=1 ' )`

The dynamic analyzer will check the syntax *w.r.t* the grammar depicted before. In this case the `INSERT` statements parse successfully.

eid	login	pass
1	Admin	Admin
2	user	1234
3	O'Brian	user1234

eid	ename	B_ sal	GP
1	Raja	1600	350
2	Tapan	800	280
3	Tamal	2600	500

(a) Database (Tables "`loginfo`" and "`salinfo`")

1. start;
2. Stmt = DriverManager.getConnection(" jdbc:mysql://172.16.1.16:1115/ demo", "scott", "tiger").createStatement();
3. Resultset rs1 = Stmt.executeQuery("`SELECT eid FROM loginfo WHERE login = ' "+ slogin +" ' AND` pass = ' "+ spass +" ' ");
4. Resultset rs2 = Stmt.executeQuery("`SELECT ename FROM salinfo WHERE B_sal> GP + ' "+vsal+" ' `");
5. Resultset rs3 = Stmt.executeQuery("`SELECT ename FROM salinfo WHERE GP > ' "+ com +" ' *` ' "+ rate +" ' ");
6. stop;

(b) Web application

1. start;
2. Stmt = DriverManager.getConnection(" jdbc:mysql://172.16.1.16:1115/ demo", "scott", "tiger").createStatement();

3a. String str = "`INSERT INTO INPUT ( GID, IID, u_input) VALUES ('3', '1', ' "+slogin+" ' )`"
int i = check(str);

3b. String str1 = "`INSERT INTO INPUT ( GID, IID, u_input) VALUES ('3', '2', ' "+spass+" ' )`"
i = check(str1);

3. Resultset rs1 = Stmt.executeQuery("`SELECT eid FROM loginfo WHERE login= (SELECT u_input FROM INPUT` WHERE GID = 1 AND IID = 1) AND pass = (SELECT u_input FROM `INPUT` WHERE GID = 1 AND IID = 2) ") ;

4a. String str2 = "`INSERT INTO INPUT ( GID, IID, u_input) VALUES ('4', '1', ' "+vsal+" ' )`"
i = check(str2);

4. Resultset rs2 = Stmt.executeQuery("`SELECT ename FROM salinfo WHERE B_sal> GP + (SELECT u_input FROM INPUT` WHERE QID = 2 AND IID = 1)");

int mul = rate * com ;

5a. String str3 = "`INSERT INTO INPUT ( GID, IID, u_input) VALUES ('5', '1', ' "+mul+" ' )`"
i = check(str3);

5. Resultset rs3 = Stmt.executeQuery("`SELECT ename FROM salinfo WHERE GP > (SELECT u_input FROM INPUT` WHERE QID = 3 AND IID = 1)");

6. stop;

(c) Modified web application

QID	IID	u_input
3	1	user
3	2	x'OR1=1
4	1	1'OR1=1
5	1	200

(d) Dynamic phase (Table "`INPUT`")

Fig. 1: An example

Observe that, the execution of inner queries select the inputs from "Input" table and yield the following:

```
SELECT eid FROM loginfo WHERE login = user AND pass = x'OR 1 = 1
```

As there is no credential (user and x'OR1=1) in the table "loginfo", the execution finally results into "NO ROW SELECTED".

4.2 Encoding-based Approach

As we know any modification of query by attacker-inputs using concatenation operation is only possible if both the query languages and the input strings use same alphabet. What if we change the alphabet of input strings different from the SQL? This mitigates the vulnerable effect of inputs when directly concatenated to the original SQL statements.

Let U be the alphabet of user inputs which may overlap with the alphabet of SQL characters. Let Σ_{new} be a new alphabet different from SQL characters. We define the following function F which encodes any symbol of U into a string of Σ^*_{new}:

$$F \ : \ \mathtt{U} \rightarrow \Sigma^*_{new}$$

and component-wise distributive property

$$F(x_1x_2 \ldots x_n) = F(x_1)F(x_2) \ldots F(x_n)$$

As a trivial example, let us consider the binary alphabet $\Sigma_{new} = \{0, 1\}$. Given a string "abc", the corresponding binary encoded representation of it in Σ_{new} is obtained by

$$F(abc) = F(a)\ F(b)\ F(c) = 011000010110001001100011$$

As an example, Consider the following query:

```
"SELECT eid FROM loginfo WHERE login=' '' +Slogin+" ' AND pass=' '' +Spass+" ' ''
```

Consider the inputs of Slogin and Spass are: ' OR 1=1 −− and password respectively. The encoded version of this inputs are:

$$F(\texttt{' OR 1=1 --}) = 01010101010101010101010011001010001$$
$$F(\texttt{password}) = 10010111010101001010000101001010101$$

Instead of merging the original inputs to the original query, we merged the encoded representation in the language different from the SQL characters, as depicted below:

```
SELECT eid FROM loginfo WHERE login ='01010101010101010101001100101000 1' AND
                                pass = '01010101010101010101001100101000 1'
```

Observe that, the semantics of Q is not changed due to merging of encoded inputs, leading to a mitigation of the injection attacks. However, the issue is that WHERE condition produces false positives as values are in a new language.

Extending encoding-based approach to database applications. We now extend the encoding-based approach to the case of database applications, and we discuss the following two major issues:

1. **Data storage and string comparison:** Consider the table "`loginfo`" in Figure 1(a). Suppose the administrator wants to login by providing `login= Admin` and `pass = Admin`. Appending inputs in the encoded form results into the following:

```
SELECT eid FROM loginfo WHERE login ='01100001011101010101101100101000 1'
                              AND pass = '01100001011101010101101100101000 1'
```

   where F(`Admin`)=01100001011101010101101100101000 1. When string matching operation is performed, as an enhancement, we adopt either of the followings:
   (a) *Encoded databases:* To enable the matching operation, we store encoded values of strings in the databases rather than actual strings, and we decode it to its original form after performing all operations at encoded-level. The encoded version of the table "`loginfo`" (in Figure 1(a)) is

login(varbinary)	Pass(varbinary)
01100001011101010101101100101000 1	01100001011101010101101100101000 1
10010010010101001010010100001 0	10101010110010101100101010100 0

   Observed that, encoded formed of database may occupy more storage space than the original one, and may take more time on performing database operations. Therefore, the technique might be suitable for small database systems.
   (b) *Conversion on-the-fly:* Storing values into the database in raw binary format does not seem convincing *w.r.t*. time and space complexity point of view, and it can take too much overhead in decoding. As a remedy, the database data can only be encoded whenever required by the quires to compare with the encoded inputs. Shown in the following example.

```
"SELECT eid FROM loginfo WHERE databaseEncode(login)='"+
applicationEncode(Slogin)+"'AND databaseEncode(pass)='"+applicationEncode(Spass)+"'"
```

   Observe that the encoding functions `databaseEncode` and `application Encode` are implemented at database layer and application layer respectively.

2. **Transformation of traditional string operations into semantically equivalent operations in the new language:** We want our encoding function such that all the string operation in the encoded domain should provide same results as in the original domain. This can be achieved through homomorphism property. Homomorphic encoding allows specific types of computations on the encoded texts such that the generated result matches with the result of the same operation on the original text. Observe that, traditional

substring matching operations may not satisfy homomorphic property in the proposed encoded domain. Suppose we want to search substring BH in another string CDE. The encoded form is shown in Figure 2. Note that, this encoding will result an incorrect matching. To solve this using homomorphic

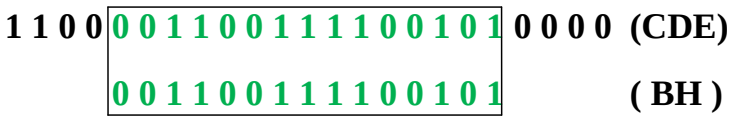

Fig. 2: Incorrect matching of encoded strings

property, we propose the following two possible enhancements:

(a) *Delimiter based encoding:* We now redefine the encoding function by introducing delimiter at the Boundary. For example,

$$F(CDE) = 01000011; 01000100; 01000101$$

In this example, the new alphabet has three symbols 0, 1 and ; . As these three symbols can not uniquely represented using single bit, the encoding function represents 0, 1, ; by 00, 11 and 010 respectively, as:

$F(CDE) = 00110000000011110100011000000110000010001100000011001 1$

(b) *Use pre-defined procedures at database layer:* We may also use any of the pre-defined procedure into database library such as Binary, Hex, etc. This ensures that matching is done on the byte boundary. For example,

```
"SELECT eid FROM loginfo WHERE BINARY(login)='" +BinEnc(Slogin)+ "' AND
                              BINARY(pass) = '" +BinEnc(SPass)+ " ' "
```

or

```
"SELECT eid FROM loginfo WHERE HEX(login)='" +HexEnc(Slogin)+ "' AND
                              HEX(pass) = '" +HexEnc(SPass)+ " ' "
```

Where `HEX` and `BINARY` functions are predefine in database library. `Hex` converts strings into hexadecimal and `BINARY` convert strings into binary form. `BinEnc` and `HexEnc` are the encoded functions defined at application level.

Observe that, the homomorphic property can easily be extended in case of other operations as well, e.g. `LIKE`, `GROUP BY`, etc.

4.3 Assertion-based Approach

Instrumenting programs by adding assertions at critical positions of the programs is one of the most successful program verification approaches, which has various application domains *e.g*. Safety property checking, Proving program correctness, Automatic test-case generation and fuzzing, Proof carrying code, Input filter generation, etc. [5].

Program verification has recently received renewed attention from the Software Engineering community. One very general reason for this is the continuing and increasing pressure on industry to deliver software that can be certified as safe and correct. A more specific reason is that program verification methods fit very naturally the so-called Design by Contract methodology for software development, with the advent of program annotation languages like JML [13].

It is common for assertions to be defined as a super-set of boolean expressions, since they may have to refer to the values of expressions in the current state of the program. If exactly the same syntax is used for assertions and boolean expressions, it will be easier for ordinary programmers to write specifications.

Assertions that hold before and after execution of a program, preconditions and postconditions respectively, will allow one to write specifications of programs or Hoare Triples. The intuitive meaning of a specification {P} C {Q} is that if the program C is executed in an initial state in which the assertion (precondition) P is true, then either execution of C does not terminate or, if it does, assertion Q (a postcondition) will be true in the final state. As example, {a > 3} a = a + 7 {a>10} represents that, for any state satisfying a > 3, the execution of a=a+7 will end with satisfying a > 10.

We extend the assertion based program verification to detect and prevent SQLIA. The approach consists of the following phases:

1. Preprocessing the input strings by filtering meta-characters.
2. Identify l_i: Q_i ($\{I_{ij} \mid j = 1, \ldots, m\}$)
3. Insert assertion after each l_i which checks the correctness of Q_i according to the specification.

Observe that assertion can be implemented either at database level or at application level. An application level assertion is illustrated in the following example: Consider a web application which contains the following:

```
rs = "SELECT eid FROM loginfo WHERE login='"+Slogin+"' AND pass='"+Spass+" ' "
```

As bad inputs may change the semantics of the above query, our approach will add a piece of code (assertion) in the web application which checks the content of `rs` during run-time in order to verify that no SQLIA happens. Below is a sample application-level assertion checking code:

```
int flag = 0 ;
while (rs.next) {
      if (!Slogin.equals(rs.getString(1)) || !Spass.equals(rs.getString(2))) {
            flag = 1 ;
            break ;
      }
}
if (flag == 1)
      print (SQL Injection attack);
```

We may also implement the assertion-based checking at database level. Observe that, this technique introduces much computational overhead in case of large database.

5 Discussion And Complexity Analysis

The complexity of Query Rewriting approach mainly depends on (*i*) Parsing, (*ii*) Data Insertion and (*iii*) Nested query execution on "`Input`" table. The worst-case time complexity of LALR parser is linear *w.r.t.* the number of inputs. Considering less number of inputs in practice, this approach does not introduce much overhead at all. The complexity of second approach depends on the encoding domain. The third approach introduces a high-overhead in computational complexity because of runtime data checking. Overall, query rewriting-based approach is suitable for web application containing simple queries with few inputs and few nested forms. The encoded-based technique is suitable for web application involving few operations on database data. The third approach is suitable for small database web applications. Below, in table 1, we provide a comparative analysis of our proposed approaches *w.r.t.* the existing ones. In the last column, we denote by n the number of SQL queries in the application.

Technique	Auto-detection	Auto-prevention	Identification of all input sources	False positive	False negative	Modify code base	Time complexity
Java Static Tainting [15]	√	x	√	x	√	x	O(n)
Security Gateway [19]	x	√	x	x	x	x	O(n)
WebSSARI [12]	√	x	√	x	x	x	O(n)
SQLrand [2]	√	√	x	√	√	√	O(n)
SQL DOM [16]	N/A	√	N/A	x	√	√	O(n)
IDS [22]	√	x	√	√	√	x	O(n)
JDBC-Checker [7]	√	x	x	x	x	x	O(n)
AMNESIA [9]	√	√	√	√	√	x	$O(2^n)$
CANDID [1]	√	√	√	√	√	x	O(n)
CIAOs [18]	√	√	√	x	x	x	O(n)
DIGLOSSIA [20]	√	√	x	√	x	x	O(n)
Query Rewriting	√	√	√	x	x	√	O(n)
Encoding-based	√	√	√	x	x	x	O(n)
Assertion-based	√	√	√	x	x	x	$O(n^2)$

Table 1: Comparison of proposed techniques *w.r.t.* existing ones in the literature

6 Conclusion

This paper proposes three new approaches to detect and prevent SQL Injection Attacks. The proposals can be treated as an alternative solutions *w.r.t.* the existing ones in the literature. This paper also described the advantages and shortcoming of each proposed technique *w.r.t.* applicability point of view. We are now in process of implementing tools based on the proposals.

References

1. Bisht, P., Madhusudan, P., Venkatakrishnan, V.N.: Candid: Dynamic candidate evaluations for automatic prevention of sql injection attacks. ACM Trans. Inf. Syst. Secur. 13(2), 14:1–14:39 (2010)
2. Boyd, S.W., Keromytis, A.D.: Sqlrand: Preventing sql injection attacks. In: In Proc. of the 2nd ACNS Conference. pp. 292–302 (2004)
3. Buehrer, G., Weide, B.W., Sivilotti, P.A.G.: Using parse tree validation to prevent sql injection attacks. In: Proc. of the 5th International Workshop on SEM. pp. 106–113. ACM (2005)
4. Clarke, J.: SQL Injection Attacks and Defense. Syngress Publishing, 1st edn. (2009)
5. Comini, M., Gori, R., Levi, G.: Assertion based inductive verification methods for logic programs. Electr. Notes Theor. Comput. Sci. 40, 52–69 (2000)
6. Cook, W.R.: Safe query objects: statically typed objects as remotely executable queries. In: In Proc. of the 27th ICSE. pp. 97–106. ACM (2005)
7. Gould, C., Su, Z., Devanbu, P.: Jdbc checker: A static analysis tool for sql/jdbc applications. In: Proc. of the 26th ICSE. pp. 697–698. IEEE Computer Society (2004)
8. Halder, R., Cortesi, A.: Obfuscation-based analysis of SQL injection attacks. In: Proc. of the 15th IEEE Symposium ISCC. pp. 931–938. IEEE (2010)
9. Halfond, W.G.J., Orso, A.: Amnesia: Analysis and monitoring for neutralizing sql-injection attacks. In: Proc. of the 20th IEEE/ACM ASE. pp. 174–183. ACM (2005)
10. Halfond, W.G.J., Orso, A.: Combining static analysis and runtime monitoring to counter sql-injection attacks. SIGSOFT Softw. Eng. Notes 30(4), 1–7 (July 2005)
11. Halfond, W.G., Viegas, J., Orso, A.: A Classification of SQL-Injection Attacks and Countermeasures. In: Proc. of the IEEE International Symposium on Secure Software Engineering. IEEE (2006)
12. Huang, Y.W., Yu, F., Hang, C., Tsai, C.H., Lee, D.T., Kuo, S.Y.: Securing web application code by static analysis and runtime protection. In: Proc. of the 13th International Conference on WWW. pp. 40–52. ACM (2004)
13. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of jml: A behavioral interface specification language for java. SIGSOFT Softw. Eng. Notes 31(3), 1–38 (2006)
14. Lin, J., Chen, J., Liu, C.: An automatic mechanism for adjusting validation function. In: 22nd AINA, 2008, Okinawa, Japan. pp. 602–607. IEEE Computer Society
15. Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in java applications with static analysis. In: Proc. of the 14th Conference on USENIX Security Symposium - Volume 14. pp. 18–18. USENIX Association (2005)
16. Mcclure, R.A., Krger, I.H.: Sql dom: compile time checking of dynamic sql statements. In: in ICSE05: Proc. of the 27th ICSE. pp. 88–96. ACM (2005)
17. Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., Evans, D.: Automatically hardening web applications using precise tainting. In: Security and Privacy in the Age of Ubiquitous Computing, IFIP TC11 20th International Conference on SEC, 2005. pp. 295–308

18. Ray, D., Ligatti, J.: Defining code-injection attacks. In: Proc. of the 39th POPL. pp. 179–190. ACM (2012)
19. Scott, D., Sharp, R.: Abstracting application-level web security. In: Proc. of the 11th International Conference on WWW. pp. 396–407. ACM (2002)
20. Son, S., McKinley, K.S., Shmatikov, V.: Diglossia: detecting code injection attacks with precision and efficiency. In: Proc. of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 1181–1192. ACM (2013)
21. Su, Z., Wassermann, G.: The essence of command injection attacks in web applications. In: Conference Record of the 33rd POPL. pp. 372–382. ACM (2006)
22. Valeur, F., Mutz, D., Vigna, G.: A learning-based approach to the detection of sql attacks. In: Proc. of the 2nd International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 123–140. Springer-Verlag (2005)