Indonesian Journal of Electrical Engineering and Informatics (IJEEI)
Vol. 8, No. 2, June 2020, pp. 364~377
ISSN: 2089-3272, DOI: 10.11591/ijeei.v8i2.1107

Web Application Reinforcement via Efficient Systematic Analysis and Runtime Validation (ESARV)

Zeinab Lashkaripour
Department of Computer Engineering, Faculty of Engineering, Velayat University, Iranshahr, Iran

Article Info

Article history:
Received Mar 14, 2019
Revised May 2, 2020
Accepted May 5, 2020

Keywords:
Systematic analysis
Runtime validation
ESARV
Web application
Security
SQLIA

ABSTRACT

Securing the data, a fundamental asset in an organization, against SQL Injection (SQLI), the most frequent attack in web applications, is vital. In SQLI, an attacker alters the structure of the actual query by injecting code via the input, and gaining access to the database. This paper proposes a new method for securing web applications against SQLI Attacks (SQLIAs). It contains two phases based on systematic analysis and runtime validation and uses our new technique for detection and prevention. At the static phase, our method removes user inputs from SQL queries and gathers as much information as possible, from static and dynamic queries in order to minimize the overhead at runtime. On the other hand, at the dynamic phase, the prepared information alongside our technique are used to check the validity of the runtime query. To facilitate the usage of our method and show our expectations in practice, ESARV was implemented. The empirical evaluations demonstrated in this paper, indicate that ESARV is efficient, accurate, effective, and also has no deployment requirements.

Corresponding Author:
Zeinab Lashkaripour,
Department of Computer Engineering,
Faculty of Engineering, Velayat University,
Iranshahr, Iran.
Email: z.lashkaripour@velayat.ac.ir

1. INTRODUCTION

Web applications are widely used due to providing accessibility and convenience. Their vast usage makes them a suitable target for an attacker; therefore preserving their security becomes essential. Despite the significance of web application security, less attention has been given in this area and the reasons can be: they are written by developers that have less programming and security skills, some web applications are produced by integrating works from several developers, so reviewing and verifying the code completely is not always possible, and finally, the developers are asked to focus on functionality rather than security; therefore we might have insufficient input validation [1]. As a result, although we have been given security patterns and guideline such as the ones introduced in [2] that can help standardize the design and development of the security architecture, but unsecure web applications still exist. Furthermore, security experts could also use Honeypots [3] for monitoring various attacks in web applications and collecting information in order to boost security.

Different types of attacks exist for web applications and according to OWASP Top Ten in 2017 [4], SQLIA has the highest frequency among them all. This shows the significance of securing web applications and their data against this frequent attack. SQLI occurs when an attacker attempts to inject code via the input and change the semantic of the intended query. SQLIAs have different types: tautology, illegal/logically incorrect, union, piggy backed, blind injection, timing attacks, alternate encoding and Stored Procedure (SP). Here we will give an example of tautology (for more information please refer to [5]-[6]). Consider a login

query with two fields that the attacker inserts "' or 1=1 --" into the user field and nothing for the pass field; therefore the resulted query is:

*SELECT * FROM UserAccounts WHERE user='' or 1=1-- ' AND pass=''*

The result of the above injection is a query with a WHERE clause that is always true, thus all the data in the UserAccounts table is retrieved. As mentioned earlier, the reason is that in case of insufficient input validation the attacker is capable of changing the semantic and structure of the query. Intentions of SQLI could be identifying injectable parameters, bypassing authentication, determining the database scheme, extracting/adding/modifying data, performing Denial of Service (DoS), evading detection, performing privilege escalation, and downloading/uploading files [7]. As a result, the consequence of this security breach is unrestricted access to the database which is a fundamental asset in any organization.

Presenting an approach against SQLIAs that is accurate, effective, and efficient at the same time and requires no deployment requirements has been an issue. It has always been a trade-off between these factors; for instance, when preserving effectiveness and precision, efficiency on the other hand is decreased. The difference between these factors has been noticeable in previous works; thus, we are introducing a method that will overcome this issue in the best form possible. As a result, we have proposed a method which is a combination of systematic analysis and runtime validation. The initial form of our proposed detection and prevention technique was introduced in [8], semi-automatically (although incomplete) extended in [9], and finally, after overcoming the shortcomings of our previous works and completing our method, ESARV was resulted to facilitate the method's application. Conceptual advantages of our method in comparison to previous works are presenting a simple and effective detection and prevention technique, requiring the least runtime processing, and using our proposed policy for identifying input locations. Furthermore, to the best of our knowledge, using reaching definition analysis (Use-Def (UD) chains) as part of the information extraction process, for the first time. Finally, considering all types of queries (static and dynamic) and inputs (string and numeral) to protect the web application and its database from unrestricted access. Moreover, the proposed method has no deployment requirements and is testified to be precise, effective, and have a negligible overhead at runtime.

The main contributions of this work are summarized as follows:

- Presenting a new combinational method that overcomes SQLIAs by combining systematic analysis and runtime validation. Our method is accurate, and has no false positives or false negatives.
- Using UD chains for information extraction in the static phase for the first time in this field.
- Using our new detection and prevention technique to remove user inputs from SQL queries, and gather the required static models and input locations from various query types.
- Implementing our method in a tool, ESARV, for Java-based web applications. It requires no manual code modification or additional infrastructure and performs detection and prevention automatically. ESARV performs a major part of the required processing at the static phase and has no overhead at runtime for these actions. As a result, the negligible overhead leads to high performance.

The remainder of this paper is organized as follows: section 2 introduces related works; the next section discusses our proposed method and its specifications in detail. Extensive practical results and discussions are illustrated in section 4 and finally, conclusion is given in the last section.

2. RELATED WORK

In this section, we have divided the related works into three categories: static, dynamic and combinational. Each of these categories are introduced along with their characteristics.

2.1. Static

These techniques have no runtime overhead and can be used before application's deployment. They help developers identify the vulnerabilities in order to reduce and/or remove them and gain more reliability. Despite their advantages, they have some shortcomings: manual alteration of the vulnerable parts which is tedious and time consuming, not being successful in SP attacks [10] and not paying attention to dynamic queries because their full structure will not be specified until runtime [6].

In SQL DOM [11] and Safe Query Objects [12] the process of creating a query is performed systematically which uses a type checking Application Programming Interface (API) in order to make the database access secure and reliable. On the other hand, they are expensive for legacy code and demand learning a new API [5].

Penetration testing tools such as MySQLInjector [13], V1p3R (Viper) [14], Sania [15], SAFELI [16], WAVES [17], [18], and [19] gather information from the web application and in order to analyze the application's response, they inject attacks according to the information gathered. V1p3R uses stored error patterns, and Sania uses SQL parse tree comparison for SQLIA detection. While MySQLInjector, [18], and

[19] output the results of the attacks, SAFELI generates test cases according to the constructed abstract syntax tree of each hotspot (a spot in a web application that has interaction with the underlying database). Success in these tools depends on the completeness of the injected attacks, whereas no web application modification is required. SecuBat [20] is also a web application scanner that identifies SQLI and Cross-Site Scripting (XSS) vulnerabilities; therefore it is suitable for securing the web application.

Prepared statements, if properly used can also be a solution to reduce SQLIAs and have no overhead for dynamic statement analysis. These parameterized statements, are prepared SQL templates executed with high efficiency. SecurePHP [21], [22], and SQLPIL [23] attempt to find the vulnerable queries and use prepared statements instead. While [22] and SQLPIL automatically replace the vulnerable queries with the safe ones, SecurePHP requires the developer's effort.

2.2. Dynamic

Dynamic techniques perform all the demanded operations at runtime. The introduced techniques consider dynamic queries and use a runtime generated model for SQLIA detection, while having the overhead of runtime model generation.

SQLGuard [24], CANDID [25], and DSD [26] compare the actual and runtime parse trees at runtime so that in case of mismatch (SQLIA), the query would not be executed. SQLGuard partially covers dynamic queries, and its shortcomings are not being capable of identifying SP attacks [10] and the need for code modification. On the other hand, CANDID requires no manual code modification and partially identifies the attacks. Finally, DSD demands no access to the source code, while having a low false positive rate [26].

2.3. Combinational

These techniques contain two phases named static and dynamic. Although the operations fulfilled in the static phase have no overhead at runtime, success depends on the accuracy of this phase.

SQLrand [27] is based on randomization and appends a key to SQL standard keywords in the static phase. Since the key is not known by the attacker SQLIAs are identified. At runtime, the proxy de-randomizes the query and if all the keywords contain the key, the query is sent with standard keywords to the database for execution. Advantages of SQLrand are performing de-randomization inside the proxy and hiding the database errors via the proxy. In contrast, the security of SQLrand depends on the security of the key and it's not capable of identifying illegal/logically incorrect, SP, and alternate encoding attacks [5].

SQLiGoT [28] and SQLiDDS [29] have a training phase; therefore their accuracy depends on the percentage of training samples and fine tuning of the training parameters. Using graph of tokens and Support Vector Machine (SVM), SQLiGoT normalizes the queries into a sequence of tokens and generates a weighted graph from the tokens. After that, it trains an SVM classifier using the centrality measure of nodes and finally at runtime, malicious queries are identified by means of the classifier. SQLiDDS requires no source code access and uses clusters of injected structures for identifying SQLIAs. In the offline phase the collected SQLIAs are transformed into a text form so that document similarity measurement is simplified. Then, they are grouped into clusters based on their document similarity and attack vectors in each cluster are merged into a document. At runtime, SQLiDDS detects the attacks by comparing document similarity of an incoming query with these documents according to rejection and suspicious thresholds.

Model construction techniques also lie in the combinational group. In the static phase, a model indicating all the valid queries of each hotspot is created and at the second phase, the runtime queries are examined to see whether they match their corresponding model or not. If not, the query would not be executed and SQLIA is prevented. SP attacks and dynamic queries are an issue in these techniques. The models used in AMNESIA [30], SQLCHECK [31], [32], SQLProb [33] and [34] respectively are Non-Deterministic Finite Automaton (NDFA), augment queries with key-marked inputs, SQL-graph, parse tree, and attribute removed queries. AMNESIA adjusts the web application by inserting a call to the runtime monitor before the query execution. After that, if the automaton accepts the runtime query, it would be executed. In SQLCHECK valid queries are those parsed by the augment grammar and sent without the keys to the database for execution. SQLCHECK's security depends on the security of the key. Furthermore, it requires manual code modification for inserting keys in the queries; therefore incompleteness is an issue. [32] requires no code modification which will spare money and time. Furthermore, by using SQL-graph only the queries that are supersets of other queries are inspected for compatibility of their static and dynamic SQL-finite state machines. It should be mentioned that [32] has also used parallel implementation in order to decrease runtime execution. SQLProb extracts inputs at runtime so that they could be validated by the output of the Parse Tree Generator, and executed if normal. SQLProb requires no code modification, is independent of the web application's programming language, and is capable of identifying all SQLIAs. On the other hand, it's limited to Mysql database [35] and the main causes of delay are query alignment, parse tree generation,

and user input validation. Finally, authors in [34] use the opinion of Removing Attribute Values (referred to as RAV). For SQLIA detection, the two extracted models are compared via XOR and if the result of the comparison is zero the query is safe to be executed. Simplicity is the advantage of this method, while its disadvantage is performing unnecessary inspections at runtime which leads to overhead increase.

Taint propagation is another technique that can be used to secure web applications. The main idea is that any variable which can be changed by the user is tainted and if used in execution of sensitive functions, a security breach may occur [36]. Here we will introduce three of these techniques that are capable of identifying all types of SQLIAs. WebSSARI [37] uses static analysis for inspecting tainted flows against preconditions of sensitive functions. During the analysis, WebSSARI suggests sanitization functions for those points where the preconditions are not satisfied and automatically inserts runtime guards in the vulnerable parts of code. Although being effective, WebSSARI has some disadvantages, it requires some information provided by the developer [38] and cannot remove SQLI vulnerabilities, but only lists the inputs as black or white [39]. WASP [40] and IDL [41] are based on positive tainting and perform automatic syntax aware validation. That is, the query is tokenized into a sequence of SQL keywords, operators, and literals. Then if all of the tokens except the literals are made from trusted data, the query is safe for execution. WASP and IDL require transformation, the former on the bytecode and the latter on the source code. WASP demands to specify trusted external data sources because they are not hard coded in the application and if not specified, false positives occur. Furthermore, at runtime IDL compares the query result size of the runtime and actual query via the usage of Equivalence and Largest Selectivity algorithm [42]. IDL requires developer intervention in some conditions and generates false positives in queries with multiple tables or very high complex "WHERE" clause.

3. PROPOSED METHOD

Our proposed method is implemented in a tool named ESARV (Figure 1), to automate all the operations, and facilitate its usage.

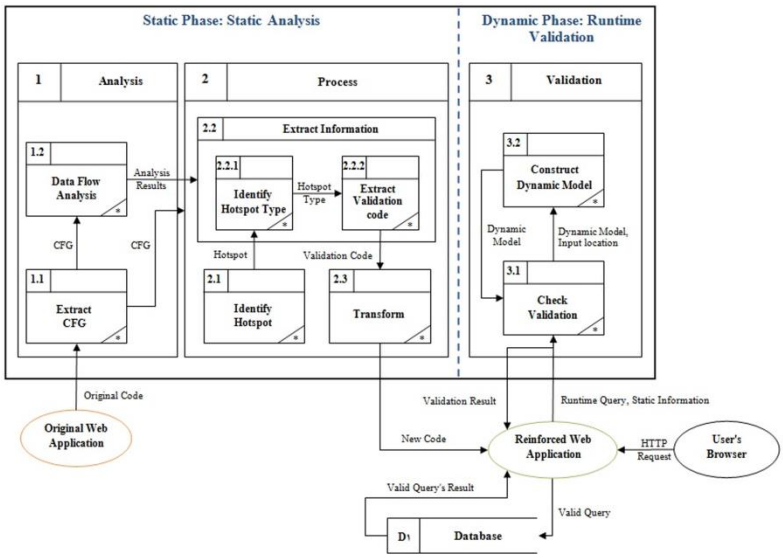


Figure 1. Architecture of ESARV [43]

As shown in Figure 1, ESARV is capable of reinforcing Java-based web applications and their underlying databases against SQLIAs. This tool is based on systematic analysis and runtime validation and contains two phases named static and dynamic. The main components of ESARV are Analysis, Process, and Validation. The Analysis component is an introduction to achieve the main objective of our method in the other two components that perform the basic task. The input of ESARV is the original web application that enters the Analysis component and the output is the reinforced form of the input which exits the Process component. The overall goal in the static phase is generating the reinforced form of the original web application, which requires constructing models and gathering information for the latter phase that aims to detect and prevent SQLIAs.

The static phase which is in charge of systematic analysis contains the Analysis and Process components and its operations can be summarized as follows:

1. Generating the Control Flow Graph (CFG) to indicate the structure of the code.

2. Performing Data Flow Analysis (DFA) on the output of the previous step in order to extract the required information.
3. Identifying hotspots within the code and determining their types.
4. Processing the information of the last two steps and extracting validation code.
5. Reinforcing the original web application by inserting validation code in proper locations.

On the other hand, the dynamic phase is in charge of runtime validation and contains the Validation component in which both validation and dynamic model construction will occur. The operations of this phase, as stated below, are repeated until either an SQLIA is detected or no attack has taken place and the runtime query is safe for execution. These operations can be summarized as follows:

1. Checking the input location based on the information gathered in the static phase.
2. If the result of the previous step is positive or in other words, the query is valid up to now, the dynamic model is gradually constructed by means of our new technique.

Each of the main components in ESARV would be explained in detail in this section.

3.1. Analysis

The Analysis component performs the first two steps of the static phase mentioned before. This main component is used for extracting information for our detection and prevention technique. The symbols used in this section are shown in Table 1.

Table 1. Symbols used in reaching definition analysis

Symbol	Definition
Entry	The entry point in the CFG
S	A basic block in the CFG
P	A member in the set of predecessors (comes before S in the CFG)
d	A definition in the UD chain
y	A variable
DEFS[y]	The set of all definitions that are assigned to `variable y

As we can see in Figure 1, the Analysis component contains two subcomponents named Extract CFG and DFA. For both of these subcomponents we have used the facility that Soot (http://www.sable.mcgill.ca/soot/) [44] provides. As mentioned earlier, our method is based on systematic analysis and should be performed on a model that is constructed from the code. For this purpose, we extracted a CFG which will be used inside the DFA section in order to gather the information required for producing the validation code (for more information on CFG and DFA please refer to [45]). Based on our goal, we have used reaching definitions for DFA. According to [45] this technique is defined as follows:

Definition1. *It is said that a definition d reaches a point p if there is a path from the point immediately following d to p, such that d is not "killed" along that path. A definition of a variable x is killed if there is any other definition of x anywhere along the path.*

Reaching definition analysis as shown in equations (1) to (5) [45] is a forward data flow framework with the data flow values of definitions.

$$OUT[Entry] = \varnothing \quad (1)$$

$$IN[S] = \cup P \in pred[S]\ OUT[P] \quad (2)$$

$$OUT[S] = GEN[S] \cup (IN[S] - KILL[S]) \quad (3)$$

$$GEN[di : y \leftarrow f(x1, \ldots, xn)] = \{di\} \quad (4)$$

$$KILL[di : y \leftarrow f(x1, \ldots, xn)] = DEFS[y] - \{di\} \quad (5)$$

In this technique, four sets are computed for each node. GEN and KILL are the two local sets which are used for computing the global IN and OUT sets. For convenience, reaching definitions can be stored in the form of UD chains, where the use (U) of a variable and all the definitions (D) that can reach that use are considered. In our method, all these UD chains should be computed and propagated until the hotspot is reached. The hotspot is a use of the related variable(s) inside the query and all the definitions that reach that variable(s) in the hotspot are the definitions that we are looking for in order to produce the validation code required for reinforcement. Using Soot, we have computed the UD chains of the corresponding variable(s) in the hotspot. These chains, which are the results of DFA, are then given to the Process component for further inspection.

3.2. Process

The Process component performs steps three to five of the static phase mentioned in section 3 and contains three subcomponents as shown in Figure 1. This basic component of the static phase is responsible for generating the reinforced form of the original web application. For this purpose, it produces the required validation code according to the CFG, analysis results, hotspot type, and our detection and prevention technique and then transforms the web application accordingly.

The first action is identifying the hotspots through traversing the CFG and this is performed by the Identify Hotspot subcomponent. In this part, locations in the CFG that contain statements like executeQuery and executeUpdate are identified as hotspots. After that, the query or the variable(s) of the related hotspot could be accessed according to the results of DFA. The policy of the Extract Validation Code subcomponent for generating the proper validation code based on our detection and prevention technique varies according to the three query types mentioned below:

1. Containing no input (constant).
2. Containing input and having a clear structure at compile time (static). In the UD chains of this type of query, each use has only one reaching definition; therefore, we can obtain the structure of the valid query and input locations by propagating the definitions.
3. Containing input and having a structure that is not clear until runtime (dynamic). In the UD chains of this type of query at least one use has more than one reaching definition; therefore different inputs, result in different queries at runtime.

One of the specifications of our proposed method is gathering as much information as possible in the static phase in order to lower the runtime overhead. In addition to the hotspot type, another important part of this information is related to input locations. These locations are detected via our new policy which is based on the symbols demonstrated in Table 2.

Table 2. Symbols used for detecting input locations

Symbol	Definition
d OR d1	A definition in the UD chain
i	An input element (either direct or indirect)
inputFunction()	Any function used for receiving an input
p	An input variable
ci	An element containing an i element on its RHS
r()	A relationship between various elements that creates the RHS of a definition
x1~xn	A query element
c OR c1~cn	A constant element (either direct or indirect)
x OR y	A variable on the Left-Hand Side (LHS) of a definition
n	An element that its information is needed for extracting input locations
nn	An element that its information is not needed for extracting input locations

We have divided the query into three elements: input, containing input and constant as follows:

1. Input: An element only containing functions related to receiving inputs that we indicate it with *i* and its definition is:
   - $d: i \leftarrow inputFunction(p)$

In this definition, *inputFunction* is a symbolic method used for receiving user inputs such as getParameter and getAttribute in JavaServer Pages (JSP).

2. Contains input: An element with an *i* element on the Right-Hand Side (RHS) either direct or indirect that we indicate it with *ci* and its definition is:
   - $d : ci \leftarrow r(x1, \ldots, i, \ldots, xn)$
3. Constant: An element that contains constants on the RHS either direct or indirect that we indicate it with *c*. A constant might be used for computing input locations and its definition is:
   - $d : c \leftarrow r(c1, \ldots, cn)$

We know that each definition in a UD chain contains a LHS and a RHS which based on the introduced elements can have different conditions where order is important. For identifying input locations, the information related to query elements could be either needed or not depending on their location; therefore we define them as follows:

1. Needed: An element which according to its position, its information is needed for extracting input locations. We indicate it with *n* and its definition for variables and constants is:
   - Variable: Variable *x* is a needed element in the states below:
     - $d : x \leftarrow r(x1, \ldots, xn)$
       $\exists\, d1 : ci \leftarrow r(x, \ldots, i, \ldots, xn)$
     - $d : x \leftarrow r(x1, \ldots, xn)$
     - $\exists\, d1 : y \leftarrow r(x, \ldots, ci, \ldots, xn)$
   - Constant: Constant *c1* is a needed element in the states below:
     - $d : ci \leftarrow r(c1, \ldots, i, \ldots, xn)$
     - $d : y \leftarrow r(c1, \ldots, ci, \ldots, xn)$
     - $d : n \leftarrow r(c1, \ldots, xn)$

2. Not needed: An element which according to its position, its information is not needed for extracting input locations. We indicate it with *nn* and its definition for constants (only a constant can be *nn*) is:
   - Constant *cn* in the first two states below and also constants *c1~cn* in the last state are *nn* elements.
     - $d : ci \leftarrow r(x1, \ldots, i, \ldots, cn)$
     - $d : y \leftarrow r(x1, \ldots, ci, \ldots, cn)$
     - $d : nn \leftarrow r(c1, \ldots, cn)$

Identifying the input locations, only requires the *n* elements, whereas preserving the query structures intended by the developer, requires all the query elements whether *n* or *nn*. This information will be used in our new detection and prevention technique with the pseudocode given in Figure 2.

```
Input:          sqlQuery
Output:         staticModel, input_Locations

int   location
String   staticModel
List   input_Locations
For each input in sqlQuery
{
        location=Identify_location(input)
        if(Is_string(location))                 // input is a String (normal or LIKE clause)
        {
                sqlQuery=Remove_value(sqlQuery, location)
        }
        else                                    // input is a number (either null or contains a value)
        {
                sqlQuery=Replace_value_with_space(sqlQuery, location)
        }
        input_Locations.add(location)           // add the input location to list
}
staticModel=sqlQuery
```

Figure 2. Pseudocode of the proposed detection and prevention technique (static phase) [43]

In addition to the queries intended by the developer and the input locations, the validation code might contain static variables and certainly a call to the validation function before the execution of the query. These transformations vary depending on the type of the query mentioned earlier in this section. A constant query has no possibility of SQLI and requires no further action due to input absence and identical static and dynamic models. On the other hand, when we have a static query with a clear structure at compile time the information given from the previous components needs to be processed. For this purpose, we use the UD chains of the variable(s) related to the hotspot, the CFG of the web application, our policy for identifying input locations, and also our detection and prevention technique; hence according to them, the validation code required for the corresponding hotspot is produced and passed to the next component for transformation. For dynamic queries which are the complicated form, more process is required. Since different inputs create diverse queries the validation code needs to be flexible in order to cover this type of queries. In this condition, beside transformations required for the hotspot (like the static form) some transformations might be required for the variables that have different values according to user inputs.

For securing the web application in the static phase, the final action of our proposed method is transforming it and producing the reinforced form. The validation code generated at the Extract Validation Code subcomponent is given to the Transform subcomponent for insertion in the related positions of the original web application. For instance, before the query is executed a condition is added and our validation method checks whether the static and dynamic models are identical based on the input locations. If so, the query would be safe for execution, while on the opposite side, SQLIA has taken place and an exception is thrown. For transformation, we have used the facility that Soot provides and reinforced the web application according to our demands. This reinforced web application which is the output of the first phase, will be used in the dynamic phase. The pseudocode of the Process component is shown in Figure 3.

```
Extract_information:

        Input:      CFG, analysisResult, hotspot
        Output:     Information

List  Information

// categorize varibles in the Use Def chain according to the query element types
List  Query_elements = Categorize_Variables(analysisResult, hotspot)
String type = Identify_hotspot_type(analysisResult, Query_elements)

// for constant hotspots no action is required
if (type == "static")
{
    // only one definition for each use
    Information = Propagate(CFG, analysisResult, Query_elements)
}
else if (type == "dynamic")
{
    // more than one definigion for at least one use
    Information = Extract_Dynamic_Validation(CFG, analysisResult, Query_elements)
}
-----------------------------------------------------------------------------------------------------------------------
Transform_web_application:

        Input:      CFG, Information
        Output:     reinforcedWebApplication

// add the definitions of the three locals needed: static query, dynamic query and input_Locations
reinforcedWebApplication = Add_basic_locals_definition(CFG)
if (static_type(Information))        // static query
{
    reinforcedWebApplication = Add_static_query(CFG, Information)
}
else                                 // dynamic query
{
    // add the definitions of the required static locals related to the dynamic variables
    reinforcedWebApplication = Add_static_locals_definition(CFG, Information)
    // add the values of the static models (related to the static query and its partials)
    reinforcedWebApplication = Add_static_models(CFG, Information)
}
reinforcedWebApplication = Add_input_locations(CFG, Information)
reinforcedWebApplication = Add_dynamic_model_and_validation_call(CFG, Information)
```

Figure 3. Pseudocode of the proposed detection and prevention technique (static phase) [43]

3.3. Validation

This component of the dynamic phase, detects and prevents SQLIAs according to the information gathered in the previous phase. The Validation component as shown in Figure 1, contains two subcomponents named Check Validation and Construct Dynamic Model.

At runtime, the HTTP request containing user inputs is sent to the reinforced web application and before the query is sent to the database for execution the Validation component processes the request to check whether its valid or not (Figure 4).

```
Input:      dynamicQuery, staticModel, inputLocations
Output      boolean          // SQLI = false, valid = true

For each location in inputLocations
{
    if (Validation_check(dynamicQuery, location)
    {
        // dynamic model construction is performed based on our proposed
        // detection and prevention technique
        dynamicQuery = Construct_dynamic_model(dynamicQuery, location)
    }
    else
    {
        return  false
    }
}
if (staticModel == dynamicQuery)
{
    return  true
}
else
{
    return  false
}
```

Figure 4. Pseudocode of the Validation component (dynamic phase) [43]

The Check Validation subcomponent uses the runtime query and the static information to check the validity of user inputs. This inspection is done for each input in the query and one of the three conditions mentioned here might occur.

1. Valid input: In this condition, the related input will be removed and the dynamic phase operations will continue.
2. Attack input inside one of the internal inputs: In this condition, the related input will also be removed. But, due to the changes made to the structure of the query, the next input will not be in its place, so before further inspections the injection will be identified. As a result, the query will not be executed, hence SQLIA is prevented.
3. Attack input inside the last input: This condition occurs when the attack is not identified in one of the internal steps. Thus, the entire dynamic model is constructed and a final check is required. For this purpose, the equality of the related static (from the static phase) and dynamic (from the dynamic phase) models is checked. As a result, due to the changes made by the attacker, the structure of the models are not identical; therefore the query is malicious and will not be executed.

The Construct Dynamic Model subcomponent is in charge of constructing the dynamic model gradually by means of our new technique. Depending on the type of the input, this subcomponent will perform one of the following methods:

1. String: The input value which is inside the single quote operators will be removed and if the input contains single quote as a value it will be removed like the other characters until the single quote operator it reached. In order to preserve the query intended by the developer, the LIKE clause inside a query requires special care, due to the fact that the user input might be placed beside a pattern. In this condition, the user input will be a portion of the string inside the single quote operator not the whole.
2. Numeric: The input value is replaced with a space that shows the location of the input and the value is removed until it contains valid characters for various types of numbers.

After introducing our proposed method thoroughly, in the next section we will demonstrate the experimental results and compare ESARV with other combinational techniques.

4. EXPERIMENT AND EVALUATION

In order to evaluate ESARV and show our expectations in practice we used the test suite of AMNESIA [30] which contains five web applications with different sizes as shown in Table 3. This table indicates the size of each web application in Line Of Code (LOC), and gives a brief description for each of them. This test suite also contains two types of datasets named attack and non-attack for each web application. To preserve the full automatic characteristic of the testbed the attacks include different patterns of SQLIAs except those that require human intervention and interpretation such as second-order injection. On the other hand, the non-attack dataset contains legitimate inputs that although containing no SQLI can cause failure in simple detection techniques.

Table 3. Applications of the test suite [30]

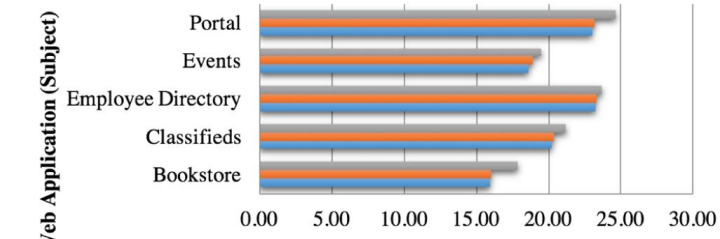
Subject	Definition	LOC
Bookstore	Online bookstore	16,959
Classifieds	Online management system for classifieds	10,949
Employee Directory	Online employee directory	5,658
Events	Event tracking system	7,242
Portal	Portal for a club	16,453

For evaluation, we have used four criterions named performance, accuracy, effectiveness, and deployment requirements. These criteria are used to compare ESARV with other combinational techniques introduced in section 2.3 (these experiments are an extention of our thesis work [43]). All of the experiments have taken place on a system which had an Intel Core i7-2600 3.4GHz with Windows 7 Ultimate SP1 OS and 8GB RAM. In this section, we will explain each criterion in detail and discuss the results thoroughly.

4.1. Performance

This criterion is used for measuring the runtime overhead. For this purpose, the non-attack dataset is used for accurate measuring of performance since maximum inspections are performed in this condition. ESARV stops whenever an injection occurs without performing further inspection and only traverses a small fraction of the query. While for RAV [34], the required inspections do not differ whether the input is an attack or not, the whole runtime query needs to be traversed. Since attackers often try to inject through the

initial inputs so that the later parts of the query are commented (by --, and /* multiline */ in MySQL and SQL Server) with no impact, RAV performs unnecessary inspections whereas ESARV speeds up detection. Figure 5 illustrates the average runtime overhead for the evaluated methods and when none of the methods is used for securing the web applications respectively. In this experiment, the non-attack dataset is used on the original web application and the time is measured by means of Apache JMeter (an open source software designed to load test functional behaviour and measure performance). After that, the same dataset is used on the web application that is secured with one of the methods under test, and again the time is measured. The difference between these two (the None row and the other two rows) is the actual overhead of the method.



	Bookstore	Classifieds	Employee Directory	Events	Portal
RAV	17.83	21.20	23.70	19.50	24.67
ESARV	16.00	20.40	23.35	18.91	23.17
None	15.97	20.23	23.25	18.58	23.04

Figure 5. Average runtime overhead [43]

As expected, Figure 5 confirms that ESARV has a lower overhead in comparison to RAV and this difference in performance would be more noticeable in cases that the queries are more complex or even more in number. In order to compare the methods accurately and specify the efficient method, we have shown the percentage of overhead and performance in Table 4.

Table 4. Performance evaluation [43]

Subject	Method	Overhead (%)	Performance Improvement (%)	Efficient Method
Bookstore	RAV	11.65	-	ESARV
	ESARV	0.19	11.46	
Classifieds	RAV	4.80	-	ESARV
	ESARV	0.84	3.96	
Employee Directory	RAV	1.94	-	ESARV
	ESARV	0.43	1.51	
Events	RAV	5.00	-	ESARV
	ESARV	1.80	3.20	
Portal	RAV	7.07	-	ESARV
	ESARV	0.60	6.47	

Maintaining the security of a web application with the least overhead is essential and between the methods under test ESARV has accomplished it. The average speed improvement in ESARV in comparison to RAV which is a simple method, is 5.32%. These results confirm that the difference in performance between ESARV and other combinational methods that are more complicated than RAV will surely be significant.

4.2. Accuracy

Accuracy is evaluated by measuring false positive and false negative (Table 5). For the former we apply the non-attack inputs to see if the method mistakenly prevents their execution, and for the latter the attacks are used on the secured web application to determine the ones mistakenly executed. The legitimate section of Table 5 shows that among the total number of legitimate inputs how many were executed correctly along with the error rate. The attack section also shows how many of the attacks were prevented safely along with the detection rate.

The results show that ESARV has no error rate in comparison to RAV and also identifies and prevents all of the attacks correctly. The first is due to the fact that our method would not prevent the execution of legitimate inputs containing the value ' (not the operator ') and consider them as non-attack inputs. On the other hand, the lower detection rate in RAV is due to not supporting numerical inputs and being dependent to input type.

Table 5. Accuracy evaluation [43]

Subject	Method	Legitimate Queries		Attack Queries	
		Allowed/Attempt (False Positive)	Error Rate (%)	Prevented/Attempt (False Negative)	Detection Rate (%)
Bookstore	RAV	552/572 (20)	3.50	3441/3473 (32)	99.08
	ESARV	572/572 (0)	0	3473/3473 (0)	100
Classifieds	RAV	513/540 (27)	5.00	3589/3589 (0)	100
	ESARV	540/540 (0)	0	3589/3589 (0)	100
Employee Directory	RAV	602/620 (18)	2.90	3921/3937 (16)	99.59
	ESARV	620/620 (0)	0	3937/3937 (0)	100
Events	RAV	828/846 (18)	2.13	3569/3605 (36)	99
	ESARV	846/846 (0)	0	3605/3605 (0)	100
Portal	RAV	981/1014 (33)	3.25	3661/3685 (24)	99.35
	ESARV	1014/1014 (0)	0	3685/3685 (0)	100

4.3. Effectiveness

A technique has a better effectiveness whenever more attacks are detected. In this section, we have compared the combinational techniques based on their capabilities against various types of SQLIAs as indicated in Table 6. This table contains three different symbols: '-' as impossible, '+' as totally possible, and '~' as partially possible. In Table 6 we have sorted the techniques so that the least supportive one is at the top of the table; thus, SQLrand, not capable in three types of SQLIAs is located at the top. AMNESIA, SQLCheck and [32] are at the next level where SP attacks are totally possible. After that SQLiGoT partially secures web applications against tautology attacks while RAV and ESARV partially identify and prevent SP attacks. Both of them are partially vulnerable to SP attack due to the fact that user defined SPs are at the database layer and the queries inside them are not available at the application server layer. It should be stated that if an application does not contain any user defined SPs, RAV and ESARV would be 100% effective. The last group, containing SQLiDDS, WebSSARI, WASP, IDL, and SQLProb can prevent all types of SQLIAs. Based on this we can classify the combinational techniques as shown in Figure 6, into three levels of effectiveness. As the colors and the name of the levels illustrate, the first level has the highest effectiveness.

Table 6. Effectiveness evaluation

Technique	Tautologies	Illegal	Union	Piggy-Backed	SP	Inference	Alternate encodings
SQLrand	–	+	–	–	+	–	+
AMNESIA	–	–	–	–	+	–	–
SQLCheck	–	–	–	–	+	–	–
[32]	–	–	–	–	+	–	–
SQLiGoT	~	–	–	–	–	–	–
RAV	–	–	–	–	~	–	–
ESARV	–	–	–	–	~	–	–
SQLiDDS	–	–	–	–	–	–	–
WebSSARI	–	–	–	–	–	–	–
WASP	–	–	–	–	–	–	–
IDL	–	–	–	–	–	–	–
SQLProb	–	–	–	–	–	–	–

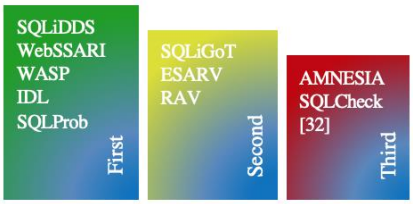


Figure 6. Effectiveness classification

4.4. Deployment Requirements

Last but not least, in Table 7 techniques are sorted according to their deployment requirements, and the best technique is located at the bottom. The considered factors are whether any code modification is required by the developer or not, if the detection and prevention are performed automatically or semi-automatically, and finally, if any additional infrastructure is required. The best techniques are ESARV and AMNESIA that require no manual code modification, perform detection and prevention automatically and finally require no additional infrastructure. According to Table 7, considering the four mentioned factors, the combinational techniques could be classified into three levels as shown in Figure 7. This figure, illustrates that the first level has the least deployment requirements.

Table 7. Deployment requirements

Technique	Manual code modification	Detection	Prevention	Additional infrastructure
SQLCheck	Needed	Semi-automatic	Automatic	Key management
SQLrand	Needed	Automatic	Automatic	Proxy server, developer learning, key management
RAV	Needed	Automatic	Automatic	Proxy server, developer learning
SQLiDDS	Not needed	Automatic	Semi-automatic	Database for injected structures
SQLiGoT	Not needed	Semi-automatic	Automatic	XML schema of all database objects
[32]	Not needed	Automatic	Automatic	Middleware (application and database)
SQLProb	Not needed	Automatic	Automatic	Proxy server
WebSSARI	Not needed	Automatic	Semi-automatic	None
WASP	Needed	Automatic	Automatic	None
IDL	Needed	Automatic	Automatic	None
AMNESIA	Not needed	Automatic	Automatic	None
ESARV	Not needed	Automatic	Automatic	None

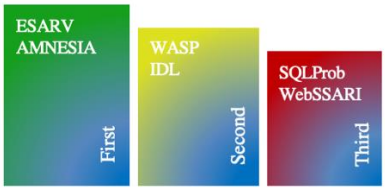


Figure 7. Deployment requirements classification

4.5. Comparative Discussion

In order to extract a final conclusion based on the evaluated criteria, further discussion is required. Therefore, the results of Figure 6 and Figure 7 require further analysis. These figures demonstrate the best options according to effectiveness and deployment requirements levels for securing a web application and the stored data against SQLIAs. AMNESIA is not a suitable choice due to its third level of effectiveness, and this would leave ESARV, WASP, IDL, SQLprob, and WebSSARI. WebSSARI, as stated in [37] has a low error rate, SQLprob has a high and IDL has a very high time complexity [41]; therefore due to the importance of precision and real time interaction, they are unsuitable. Hence, ESARV and WASP are remained and for conclusion, a more detailed comparison is performed in Table 8.

Table 8. Comparison of WASP and ESARV

Specification	WASP	ESARV
Platform specific	Java	Java
Detection	Automatic	Automatic
Prevention	Automatic	Automatic
Accuracy	100%	100%
Effectiveness	100%	100%[1]
Additional infrastructure	-	-
Manual code modification	Required	-
Simplicity	Moderate	Very high
Overhead	Low	Very low
Transformation	Bytecode (WASP) and source code (developer)	Source code

[1] If no user defined SPs exist in the web application

The first six factors of Table 8 are identical for both tools, while the remaining factors vary. ESARV is totally automatic with no manual modification, very simple, has a negligible overhead, and requires automatic source code transformation. Whereas, WASP requires manual code modification, is not as simple as ESARV, has a higher overhead than ESARV, and requires source code transformation by the developer and automatic bytecode transformation. As a result, if the web application has no user defined SPs (like the testbed) then with no doubt ESARV is the best tool for reinforcing. On the other hand, if user defined SPs exist the choice would be as follows: if effectiveness is of importance and overhead and human interaction are not an issue, WASP would be the choice. Whereas, if you are looking for an automatic solution with the least response time, ESARV is the best choice.

5. CONCLUSION

In this paper, we proposed a new method capable of securing a web application and its database against SQLIAs. This combinational method is based on systematic analysis and runtime validation and uses our proposed detection and prevention technique for information security. In the static phase, user inputs are removed from SQL queries and based on the query type, static models and input locations are gathered in

order to make the detection and prevention easier and faster at runtime. To facilitate the usage of our method and perform empirical evaluations, ESARV was implemented for Java-based web applications. Evaluations indicated that ESARV is the best choice for web application reinforcement. For this purpose, we have used the test suite of AMNESIA and also Apache JMeter for measuring performance and accuracy. ESARV was able to stop all of the attacks with no false negatives and no false positives. It also proved to be efficient by having negligible overhead for the reinforced web application. Although no manual code modification or additional infrastructure is required and detection and prevention are performed automatically, ESARV is not capable of identifying user defied SP attacks; therefore improvement is required in order to attain 100% effectiveness.

As a future work our method could be extended to support user defined SPs inside the database. Another future work would be to use our method without the demand to alter the web application's code which would lead to extra overhead, whereas at the moment modifications are performed automatically with negligible overhead. Finally, due to the significance of maintaining web application security further research is required in this field.

REFERENCES

[1] G. Lawton, Web 2.0 creates security challenges. Computer, vol. 40, no. 10, pp. 13-16, 2007, doi: 10.1109/MC.2007.367.

[2] S. Moral-García, S. Moral-Rubio, D. G. Rosado, et al., "Enterprise security pattern: a new type of security pattern," *Security and Communication Networks*, vol. 7, no. 11, pp.1670-1690, 2014, doi: 10.1002/sec.863.

[3] A. Ghourabi, T. Abbes, A. Bouhoula, "Characterization of attacks collected from the deployment of Web service honeypot," *Security and Communication Networks*, vol. 7, no. 2, pp. 338-351, 2014, doi: 10.1002/sec.737.

[4] A. Stock, Smithline N, Gigler T. OWASP Top 10-2017 RC1, 2017.

[5] W. G. J. Halfond, J. Viegas, A. Orso, "A Classification of SQL Injection Attacks and Countermeasures," *Proc. International Symp. Secure Software Engineering*, Raleigh, NC, USA, 2006, pp. 65–81.

[6] C. Song, SQL Injection Attacks and Countermeasures. M.S. diss., California State University. IEEE, 2010.

[7] Y. Gomaa, A. E. A. Ahmed, M. A. Mahmood, et al., "Survey on Securing a Querying process by Blocking SQL injection," *In 2015 Third World Conference on Complex Systems (WCCS) IEEE*, 2015, pp. 1-7.

[8] Z. Lashkaripour, A. Ghaemi Bafghi, "A Simple and Fast Technique for Detection and Prevention of SQL Injection Attacks (SQLIAs)," *International Journal of Security and Its Applications*, vol. 7, no. 5, pp. 53-66, 2013, doi: 10.14257/ijsia.2013.7.5.05.

[9] Z. Lashkaripour, A. Ghaemi Bafghi, "A security analysis tool for web application reinforcement against SQL injection attacks (SQLIAs)," *Information Security and Cryptology (ISCISC)*, 10th International ISC Conference. IEEE, 2013, doi: 10.1109/ISCISC.2013.6767326.

[10] A. Tajpour, IS. brahim, M. Sharifi, "Web Application Security by SQL Injection Detection Tools," *International Journal of Computer Science Issues*, vol. 9, no. 2, pp. 332-338, 2012.

[11] R. A. McClure, I. H. Kruger, "SQL DOM: compile time checking of dynamic SQL statements," *Proc. 27th international conference on Software engineering*, St. Louis, MO, USA, IEEE, 2005, pp. 88-96, doi: 10.1109/ICSE.2005.1553551.

[12] W. R. Cook, S. Rai, "Safe Query Objects: Statically Typed Objects as Remotely Executable Queries," *Proc. 27th International Conference on Software Engineering*, St. Louis, MO, USA, IEEE, 2005, pp. 97-106, doi: 10.1109/ICSE.2005.1553552.

[13] A. Bashah Mat Ali, A. Yaseen Ibrahim Shakhatreh, M. Syazwan Abdullah, et al., "SQL-injection vulnerability scanning tool for automatic creation of SQL-injection attacks," *Procedia Computer Science*, vol. 3, no. 0, pp. 453-458, 2011, doi: 10.1016/j.procs.2010.12.076.

[14] J. Wang, R. C. W. Phan, J. N. Whitley, et al., "Augmented attack tree modeling of SQL injection attacks," *Information Management and Engineering (ICIME)*, The 2nd IEEE International Conference, 2010, pp. 182-186, IEEE, doi: 10.1109/ICIME.2010.5478321.

[15] Y. Kosuga, K. Kono, M. Hanaoka, et al., "Sania: Syntactic and Semantic Analysis for Automated Testing against SQL Injection," *Computer Security Applications Conference*, ACSAC 2007, Twenty-Third Annual, IEEE, 2007, pp. 107-117, doi: 10.1109/ACSAC.2007.20.

[16] X. Fu, X. Lu, B. Peltsverger, et al., "A static analysis framework for detecting SQL injection vulnerabilities," *In Computer Software and Applications Conference*, COMPSAC 2007, IEEE, 2007, 1: 87-96, doi: 10.1109/COMPSAC.2007.43.

[17] Y. W. Huang, S. K. Huang, T. P. Lin, et al., "Web application security assessment by fault injection and behavior monitoring," *In Proceedings of the 12th international conference on World Wide Web*, 2003, pp. 148-159, ACM, doi: 10.1145/775152.775174.

[18] F. J. M. Vieira, Realistic Vulnerability Injections in PHP Web Applications. M.S. diss., Lisbon University, 2011.

[19] N. F. Awang, A. A. Manaf, "Automated Security Testing Framework for Detecting SQL Injection Vulnerability in Web Application," *In International Conference on Global Security, Safety, and Sustainability*, 2015, pp. 160-171, Springer International Publishing.

[20] S. Kals, E. Kirda, C. Kruegel, et al., "Secubat: a web vulnerability scanner," *In Proceedings of the 15th international conference on World Wide Web, 2006*, pp. 247-256, ACM, doi:10.1145/1135777.1135817.

[21] F. Dysart, M. Sherriff, "Automated Fix Generator for SQL Injection Attacks," *Software Reliability Engineering*, ISSRE 2008, 19th International Symp., IEEE, 2008, pp. 311-312, doi: 10.1109/ISSRE.2008.44.
[22] S. Thomas, L. Williams, "Using Automated Fix Generation to Secure SQL Statements," *Software Engineering for Secure Systems*, SESS '07: ICSE Workshops, Third International Workshop, IEEE Computer Society, 2007, doi: 10.1109/SESS.2007.12.
[23] W. Masri, S. Sleiman, "SQLPIL: SQL injection prevention by input labeling," *Security and Communication Networks*, vol. 8, no. 15, pp. 2545-2560, 2015, doi: 10.1002/sec.1199.
[24] G. Buehrer, B. W. Weide, P. A. G. Sivilotti, "Using parse tree validation to prevent SQL injection attacks," *Proc. 5th international workshop on Software engineering and middleware*, Lisbon, Portugal, ACM, 2005, pp. 106-113; doi: 10.1145/1108473.1108496.
[25] P. Bisht, P. Madhusudan, V. N. Venkatakrishnan, "CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 2, pp. 1-39, 2010, doi: 10.1145/1698750.1698754.
[26] T. Y. Wu, J. S. Pan, C. M. Chen, et al., "Towards SQL Injection Attacks Detection Mechanism Using Parse Tree," *In Genetic and Evolutionary Computing*, 2015, pp. 371-380, doi: 10.1007/978-3-319-12286-1_38.
[27] S. W. Boyd, A. D. Keromytis, "SQLrand: Preventing SQL Injection Attacks," *Applied Cryptography and Network Security*, Lecture Notes in Computer Science, Springer Berlin Heidelberg, no. 3089, pp. 292-302, 2004, doi: 10.1007/978-3-540-24852-1_21.
[28] D. Kar, S. Panigrahi, S. Sundararajan, "SQLiGoT: Detecting SQL injection attacks using graph of tokens and SVM," *Computers & Security*, no. 60, pp. 206-225, 2016, doi: 10.1016/j.cose.2016.04.005.
[29] D. Kar, S. Panigrahi, S. Sundararajan, "SQLiDDS: SQL injection detection using document similarity measure," *Journal of Computer Security*, vol. 24, no. 4, pp. 507-539, 2016, doi: 10.3233/JCS-160554.
[30] W. G. J. Halfond, A. Orso, "Detection and prevention of sql injection attacks," *In Malware Detection*, Springer US, 2007, pp.85-109, doi: 10.1007/978-0-387-44599-15.
[31] Z. Su, G. Wassermann, "The essence of command injection attacks in web applications," *SIGPLAN Not*. Charleston, South Carolina, USA, ACM, vol. 41, no. 1, pp. 372-382, 2006, doi: 10.1145/1111320.1111070.
[32] M. Muthuprasanna, K. Wei, S. Kothari, "Eliminating SQL Injection Attacks - A Transparent Defense Mechanism," *Web Site Evolution. (WSE '06) Eighth IEEE International Symp.*, IEEE, 2006, pp. 22-32, doi: 10.1109/WSE.2006.9.
[33] A. Liu, Y. Yuan, D. Wijesekera, et al., "SQLProb: a proxy-based architecture towards preventing SQL injection attacks," *Proc. 2009 ACM Symp. on Applied Computing*, Honolulu, Hawaii, ACM, 2009, pp. 2054-2061, doi: 10.1145/1529282.1529737.
[34] I. Lee, S. Jeong, S. Yeo, et al., "A novel method for SQL injection attack detection based on removing SQL query attribute values," *Mathematical and Computer Modelling*, vol. 55, no. 1, pp. 58-68, 2012, Elsevier, doi:10.1016/j.mcm.2011.01.050.
[35] R. P. Mahapatra, S. Khan, "A Survey of Sql Injection Countermeasures," *International Journal of Computer Science and Engineering*, vol. 3, no. 3, pp. 55-74, 2012.
[36] D. Ceara, M. Potet, L. Mounier, et al., Detecting Software Vulnerabilities Static Taint Analysis. University Politehnica of Bucharest, 2009.
[37] Y. W. Huang, F. Yu, C. Hang, et al., "Securing web application code by static analysis and runtime protection," *Proc. 13th international conference on World Wide Web*, 2004, pp. 40-52; ACM, doi: 10.1145/988672.988679.
[38] Y. Xie, A. Aiken, "Static detection of security vulnerabilities in scripting languages," *Proc. 15th conference on USENIX Security Symp.*, 2006, pp. 179-192.
[39] D. A. Kindy, A. S. K. Pathan, "A detailed survey on various aspects of SQL Injection: vulnerabilities, innovative attacks, and remedies," *Information Journal*, 2012.
[40] W. G. J. Halfond, A. Orso, P. Manolios, "WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation," Software Engineering, *IEEE Trans. Software Engineering*, vol. 34, no. 1, pp. 65-81, 2008, IEEE, doi: 10.1109/TSE.2007.70748.
[41] Y. S. Jang, J. Y. Choi, "Detecting SQL injection attacks using query result size," *Computers & Security*, no. 44, pp. 104-118, Elsevier, 2014, doi: 10.1016%2fj.cose.2014.04.007.
[42] A. Swami, K. B. Schiefer, "On the estimation of join result sizes," *In International Conference on Extending Database Technology*, Springer, Berlin Heidelberg, 1994, pp. 287-300.
[43] Z. Lashkaripour, A. Ghaemi Bafghi, A Tool for Detection and Persistence of Web Applications against SQLI Attacks, M.S. diss., Ferdowsi University of Mashhad, 2013.
[44] R. Vall´ee-Rai, Soot: A Java Bytecode Optimization Framework. M.S. diss., McGill University, 2000.
[45] A. V. Aho, M. S. Lam, R. Sethi R, et al., Compilers: Principles, Techniques and Tools, 2nd ed, New York: Addison Wesley, 2007.

BIOGRAPHY OF AUTHOR

Zeinab Lashkaripour received her BS and MS degree in computer engineering respectively from University of Sistan and Baluchestan, Iran and from Ferdowsi University of Mashhad, Iran. She is a lecturer in the Department of Computer Engineering, Velayat University of Iranshahr, Iran. Her current research interests are cyber security, Cloud Computing (CC), Internet of Things (IoT), and Big Data (BD).