Rule based Detection of SQL Injection Attack

Debasish Das
Department of Computer Science & Engineering
Tezpur University, Napaam(INDIA)

Utpal Sharma
Department of Computer Science & Engineering
Tezpur University, Napaam(INDIA)

D. K. Bhattacharyya
Department of Computer Science & Engineering
Tezpur University, Napaam(INDIA)

ABSTRACT

This paper presents an effective detection method RDUD for SQL injection attack. RDUD is an enhanced version of DUD [1]. The method comprises a supervised machine learning approach using a Support Vector Machine(SVM) to learn and to classify a query at runtime. Two *web profile*s - (i) legitimate web profile and (ii) attack web profile are generated for each of the *web-application* software which consists of a set of *production rules* extracted from the dynamic SQL queries. Both the web profiles are generated during training phase. At runtime a dynamic SQL query is matched with each of the *web profile* and accordingly it classify based on the matching distance. RDUD is independent of the developer's initialization of syntax rules, valid trusted string database, static or pre-generated program code checking, etc. Also the method is significant in view of its simplicity, efficient and its high detection rate in comparison to the earlier method [1].

General Terms

Algorithms, Performance, Design, Reliability, Experimentation, Security.

Keywords

web-application, SQL injection, classification, production rules, web profile, RDUD.

1. INTRODUCTION

A web-application is application software which includes dynamic web-pages such that end users can access the software through client modules that run in web browsers. The coding of client modules is in browser-supported languages such as HTML, Java, ASP, PHP etc. In three-tire web-applications, the user provides query specification as input in a pre-defined format in the front tire. These inputs are used in constructing SQL queries by the application server in the middle tire. The back tire contains the database server. Web-applications are popular due to the ubiquity of web browsers, and the convenience of using a web browser as a client, sometimes called as thin client. The ability to update and maintain web-applications without distributing and installing software on potentially thousands of client computers is a key reason for their popularity. Common web-applications include - web-mail, on-line retail sales, on-line auctions, on-line banking, and many other functional applications. Mitre's Vulnerability statistics reported in the year 2010-2011[2], points out 25 most common program errors or vulnerabilities causing most successful SQL injection attack along with other web-application attacks. These vulnerabilities are dangerous because it gives chances to the attackers to steal data from the application database. It is claimed that the SQL Injection scored highest rank among the web-application attacks. It is reported that SQL Injections are one of the most common and easiest techniques adopted by attackers to attack web servers, data servers and sometimes the network. This category of web attack is conducted for unauthorized access of web-application, breaking the role based accessibility, and violating the integrity of the data storage. A significant rise in SQL injection attack is reported by CISCO[3] too. The Information Systems Audit Cell report[4] recommends review the application control in net-banking applications by conducting penetration testing keeping in view of the prevailing guidelines by Reserve Bank of India, IT Act and other applicable regulations in India and asked to check the vulnerabilities in the applications like - SQL injection, Cross-site scripting etc. In this paper, in section 2, we report the background of SQL injection attack and its different classes. In this section we also present its existing practice of detection. We finally discuss in this section 2, the brief idea of the approach called DUD. In section 3, the propose detection approach RDUD is described. We report our problem formulation along with the detection algorithms in section 4. Experimental results and finally the concluding remarks are reported in section 5 and 6 respectively.

* The department is funded by UGC's DRS-I under the SAP

2. BACKGROUND OF SQL INJECTION ATTACK(SQLIA) AND ITS CLASSIFIC-ATION

SQLIA is an attack on web-application server. It occurs when an attacker changes the intended logic, semantics or syntax of an SQL query. The query generated dynamically based on user input, maliciously crafted with SQL keywords, operators, strings or literals execute in the database server. The structure of the query with insertion of such malicious input is different from the structure defined in the application code. Let us consider that an application database contain two attributes - *user name* and *password*. The application uses the values of the attributes for authentication, consist of the following code, written in *JSP* scripts.

query = "SELECT balance FROM account WHERE username=' " +request.getParameter("name")+"' AND PASSWORD=' "+request.getParameter("pass")+" ' ";

If a malicious input entered by a user for *username* "*jimmy*" and " ' or 'a'='a' " for *password* field, the query string becomes:

a_query=select account from users where name='jimmy' and password=' ' or 'a'='a'

Which will always be evaluated as true, and the user will bypass the authentication logic without entering the value of password field. Su and Wassermann[5] proposed a context free grammar to define syntactic formation of a valid query. Thus, a dynamically generated query is defined as legitimate if the parse tree T_q is possible based on the proposed grammar. Otherwise, the query is considered as illegitimate or SQL injected query. According to their definition, a SQL query q generated by a web-application P with the input (i_1, i_2, ..., i_n) is a malicious query causing SQL injection attack (SQLIA) if the query string *q* does not have a valid parse tree T_q with respect to the pre-defined valid grammar. In CANDID [6], a dynamically generated query based on user input referred as Candidate Query (CQ), is considered as an attack or illegitimate query if its structure does not match with the corresponding structure of the valid query. The valid query is referred as Benign Query (BQ). *BQ* is generated with valid inputs corresponding to a valid query structure. According to their[6] definition:

Definition 1: The structure of the query on any valid user input v is the control path defined in the program which is generated runtime query with the valid input data v.

Definition 2: The validity of the input can be checked by matching the runtime query structure with its respective control path and the user input data is considered as invalid if it does not match.

Turning back to our example given for the query string *query*, in the beginning of the section, the candidate input *v : s←"Jimmey OR 'a' = 'a' "* is an SQL injection attack, since it generates a query whose structure is - *select ? from ? where ?=? or ?=?* while its corresponding benign input *VR*(v) = *v*1 : s←"*John*" exercises the control path structure - *select ? from ? where ?=?*. However, in case of multiple dynamic query structure some of the conditional query structures may be similar to the attack queries. In such cases, the program accepts input value and based on condition it generates queries with different structures. In our paper we address the formulation in detecting SQL injection attack under such scenarios also. Let us consider the web program written in PHP script language shown in figure 1. In it there are three programmer intended query structures - *query1*, *query2* and *query3*. Each of the query executes based on the value of input data for *user type*. If the value of *user type* is 'G' then it executes query structure *query1*. If the value of *user type* is 'O' then it executes query structure *query2*, otherwise, it executes *query3*. Definitions of SQLIA given by [5] and [6], restrict such type of multiple query structures in applications. In our approach we propose that execution of such type of multiple query structure is justified based on the application requirement.

2.1 SQLIA Classification

SQLIA has been classified into five basic classes[1][10] with respect to the attacker's target and the vulnerabilities in web-applications. They are - (1) Bypassing Authentication(ByAut) (2) Unauthorized Knowledge of database(KDb) (3) Injected UNION Query (IUnion) (4) Injected Additional Query(AdQry) (5) Unauthorized remote execution of Procedures (UexRP). To explain the different classes of SQLIA, let us consider the PHP code written for user authentication checking shown in figure 2.

A classification of SQLIA and the attacker's achievements are illustrated in table 1.

```
(1) $connection=mysql connect();
(2) mysql select db("test");
(3) $user=$HTTP GET VARS['username'];
(4) $type=HTTP GET VARS['usertype'];
(5) $pass=$HTTP GET VARS['password'];
(6) if ($type='G') then
(7) begin
(8) $query1="select balance from account where
username = '$user' and usertype=$type" – and password
='$pass'";
(9) $result1=mysql query($query1);
end;
(10) elsif ($type='O') then
(11) begin
(12) $query2="select account from users where
usertype=$type and (username = '$user' or password
='$pass'");
(13) $result2=mysql query($query2);
(14) end;
(15) else
(16) $query3="select account from users where
username=$user and usertype=$type and password=$pass;
(17) end if;
(18) if (mysql num rows($result1)==1)
echo "Authorized without password"
(16) else echo "authorization failed";
(17) if (mysql num rows($result2)==1)
echo "Authorized with password"
(18) else echo "authorization failed";
```

Figure 1 An example web program

```
(1) $connection=mysql connect();
(2) mysql select db("test");
(3) $user=$HTTP GET VARS['username'];
(4) $pass=$HTTP GET VARS['password'];
(5) $query="select balance from account where
username = '$user' and password ='$pass'";
(6) $result=mysql query($query);
(7) if (mysql num rows($result)==1) echo "Authorized"
(8) else echo "authorization failed";
```

Figure 2 An example program for authentication checking

2.2 SQLIA Detection and Prevention: Related Work

Approaches for the detection of SQL injection attack can be categorized into - pre-generated and post-generated. Post-generated detection approaches are useful for analysis of dynamic or runtime SQL query, generated with user input data by a web application. Detection techniques under this post-generated category, executes before posting a query to the back-tire or database server. In pre-generated or static approaches programmers follow some guidelines for SQLIA detection during web-application development. An effective validity checking mechanism for the input variable data is also a requirement for the pre-generated method of detecting SQLIA.

Table 1: Classification of SQLIA

Class	Attack Query Example	Attacker's Achievement
ByAut [7][9]	Query = "select balance from account where username=" or 1=1 --' and password= ' ';	Two dashes, comment the remaining text. Expression 1=1 is always true. User will be logged in with privileges of the first user stored in the database.
KDb [7][8] [9]	Query = "select balance from account where username='prakash' and password=convert(select host from host)";	The error message consists of the database description and the name of the columns. Sometimes, it displays the table name also.
IUnion [7][9]	Query="select balance from account where username=' ' and password=' ' UNION select balance from account where acc_no='10090032' ";	The actual runtime query returns null data. However, the injected query generates data from the database.
Adqry [7][8] [9]	Query="select balance from account where username='prakash' and password=' ' ; drop table user";	The database server executes the second injected query. Thus, a harmful operation may also be performed on the database with such injected query(s).
UexRP [10][9]	Query="select account from user where username=' ' ; SHUTDOWN; and password=' ';	The third query SHUTDOWN, a stored procedure executes the scripts written on it.

[A] Post-Generated Approach

Some of the popular post-generated approach for the detection of SQLIA is found from [1]. It is found that researchers have evaluated the dynamic query strings by the - (i) Initialization of valid and invalid strings (ii) Syntax evaluation and (iii) Parse-tree grammar based evaluation. A novel approach for detection of SQLIA proposed in [6], for mining programmer intended queries with dynamic evaluation of candidate input. The idea is to construct dynamically the structure of the programmer-intended query corresponding to the runtime SQL query with candidate input. According to this approach a candidate input is considered as legitimate if the following two conditions hold:

Condition 1: Input must be benign, i.e., the candidate input must be evidently non-attacking, as envisioned by the programmer while coding the application. The query generated from the benign input is considered as a legitimate query.

Condition 2: Input must dictate the same path i.e., the structure of the programmer intended query given in the respective application program. A legitimate input to the executing program will dictate a control path to a point where query is issued. To deduce the programmer intended query structure for this particular path (i.e., control path), the candidate inputs must also exercise the same control path in the program. Given such candidate inputs, the method detect attacks by comparing the query structures of the programmer-intended query and the possible attack query. The approach suggests the need for an oracle that, given a control path in a program, returns a set of benign candidate inputs that exercises the same control path. This oracle, if constructed, may actually offer a clean solution to the problem of deducing the query structure intended by the programmer. Practically, such an oracle is hard to construct and become possible for typical application. However, Bisht and Venkatakrishnan proposes[6] a real scalable automatic solution to dynamically detect and prevent SQL injection attacks. At a more abstract level, the idea of computing the symbolic query on sample inputs seems a powerful idea that probably has more applications in systems security. Proxy-based detection method proposed in [11], consists of four different components: (i) Query selection for evaluation. (ii) Extraction of input data. (iii) Parse-tree generation and (iv) Input data evaluation. To implement such a method, a separate proxy, architecturally residing in the middle tire with the web-application server, is used for query selection, input data extraction and parse-tree generation. Based on evaluation of input data using parse-tree evaluator, if it is found normal, the query is allowed to execute, otherwise, query is considered as malicious and is not executed. Unlike, other detection technique, such technique has added advantage due to its modularity in implementation. However, detection rate depends on the algorithms applied for each of the components of the detection technique.

[B] Pre-Generated Approach

The pre-generated approaches for the detection of SQLIA, found in [1] consists of various analysis tools used for identifying the vulnerabilities. The detection approaches are mainly the mechanisms to identify the vulnerabilities in the web-application programs. These mechanisms include - (i) the validity checking and (ii) the integrity checking of web-programs. Pre-generated approaches are implemented with the software tools to check the vulnerabilities in web-programs so that the attackers cannot mis-utilize an application. By giving an application program as input to such software tools, the programmer intended SQL queries are analyzed using valid and tainted data. Based on such analysis, it reports the vulnerability points in an application program. It is also proposed the modules or methods used to rectify the vulnerability points. Detection methods proposed by the researchers and the category of implementing their approaches, and the pros and cons of each method are tabulated in the table 2. Based on our limited survey it is observed that -

(a) The pre-generated detection techniques for SQLIA depend on the effectiveness of the validity checking by the web programmer and the effectiveness of the tools applied to detect the integrity of the code that causes SQLIA.

(b) The post-generated approaches for detection of SQLIA are based on the initialization of trusted or untrusted strings, which are developer-dependent. The parse tree evaluation approach based on the pre-defined grammatical constraint, given in [5] and [6] may be considered better than the others. However, there is the possibility of manipulation of the initialized strings by the attacker, and pre-defined grammatical constraint is application dependent and restricts the variable runtime query structures.

In the next section, we briefly describe DUD, a method for dynamic SQLIA detection which attempts to overcome the shortcomings of the above methods.

2.3 DUD

DUD[1] is a post-generated approach for detection of SQLIA. It generates a master profile of legitimate dynamically generated queries and stored into a semi-structured form. At runtime, it converts the dynamic SQL query into semi-structured or XML form based on the Document Type Definition(DTD). It then matches the XML form query with the profile using algorithms for *exact matching* and *approximate matching*. At first, it executes *exact matching* algorithm and if it matches it considers the query as legitimate query and allows the query to transport to database server from execution. If it does not match, then it computes and retrieves the minimum distance with respect to all the queries available in the profile. A dynamic threshold is generated during training phase and is used for the detection of SQLIA by comparing with the distance value computed in *approximate matching*. A detailed discussion on the design, uses and its effectiveness is in [1].

3. AN EFFECTIVE RULE BASED DETECTION METHOD: RDUD

The proposed method RDUD, is an enhanced version of DUD and is a supervised machine learning classification approach. The method indexes the strings of the dynamic SQL queries and employs Support Vector Machine(SVM) to classify a runtime SQL query into normal or attack. A support vector machine (SVM) is a concept that maps the input vector into two or higher dimensional feature space (linear or non-linear mapping). SVM may be used most popularly in classification processing by analyzing data and recognizing the patterns[15]. We use standard binary linear SVM which takes a set of input data and predicts each input into two possible classes - attack or normal. By giving the computed distance value as input to SVM, it classifies into either of the two classes which makes the SVM a probabilistic binary linear classifier. The architecture of RDUD is shown in figure 3. It is composed of two major phases - (i) Training and (ii) Runtime detection;

During training phase, it performs the following activities - (i) Generation of Rule Dictionary; (ii) Generation of two web profiles(normal and attack) separately for legitimate and attack queries; The runtime phase, comprises the following steps -(i) XML-conversion of runtime SQL queries; (ii) Production rule generation; (iii) Distance computation with respect to each of the web profiles; (iv) Classification - attack or normal based on distance value; Pre-processing of dynamic SQL during training phase, comprises the generation of both the web profiles. The web profile generation consists of three basic tasks - (1) Converting a dynamic SQL query into XML form; (2) Generating production rules; (3) Writing into web profile. Depending upon the input data values, we consider that the dynamic SQL query for an application may have different structures. Also, corresponding to each legitimate structure of the SQL query, it may have multiple attack queries whose structures are different(for example see figure 1). To accommodate variable schema of both legitimate and attack queries, we convert the dynamic SQL query into XML form. In contrast to the grammar verification context, *rules*(*production*) are generated to construct *web profile* from the test input(i.e. dynamic SQL query) during training phase. Query classification at runtime starts with the extraction of production rule(PR) from the dynamic SQL query, followed by the execution of matching process with both the web profiles (attack web profile and normal web profile). Based on the matching score a query is classified into normal class or attack class.

The *PR* of a dynamic SQL query, given for classification, can be any of the following status -(i) It exists in the legitimate web profile; (ii) It exists in the attack web profile; (iii) It does not exists in both the profiles; For the first two conditions (1) and (2), it can be easily sorted out the class of the runtime query - attack or legitimate. In case the condition (3) holds, there may be two possibilities either the query itself is a new legitimate query or a new maliciously crafted attack query. In such case, the minimum distances (ω_1 and ω_2), with respect to both legitimate and attack profiles respectively are calculated. Using the value of ω_1 and ω_2, we calculate the nearest plane value(P_v). Such values are given input to the trained SVM to identify the class of the query.

At first, it retrieves the *PR* of runtime SQL query. It is then computes the two distance values(ω_1 and ω_2) using the same steps given in *Distance Computation* cited in *Training Phase* under this section. To execute the steps for computing the distance *PR* values of two dynamic SQL queries is given in algorithm 2. The algorithm is designed based on [13] calculates the distance by counting the number of unmatched rules or strings in a particular *PR* matching process. Let us consider the following two dynamic SQL queries (i) and (ii) in which both are considered as legitimate for an application.

(i) select balance from accounts with usertype=‘T’ and username=‘abc’ and password=‘p001’;

(ii) select balance from accounts with usertype=‘O’ and username=‘xyz’ -- and password=‘p001’;

Suppose, an attack query (iii) given below is -

(iii) select balance from accounts with usertype=‘T’ and username=‘abc’-- and password=‘p001’;

In this case the structure of both the queries (ii) and (iii) are similar. However, the query (iii) given above is an illegitimate or attack query. If we execute the matching process in which the dynamic structure of the SQL query includes the structure and the values then only the query can be identified as different from the legitimate query. Because in this particular application only for the usertype ‘O’, the password portion is not required. Thus, we get the minimum computed distance value is 2 when we include value portion in case of runtime query structure. To classify the above two SQL queries, we use the equation $\omega x \pm b = 0$. The plane of SVM, shown in figure 6, has two sides(+1 and -1). The value of two roots x_1 and x_2 are calculated and are plotted the points (x_1,x_2) in a two dimensions (X,Y) plane. During training phase we experimented separately with a list of legitimate and attack queries. A separating plane is drawn which separates the support vector planes into both the sides of it. The value of P_v(nearest plane value) is calculated using the equation $P_v(x) = x_1+x_2$ of each SQL query. For a given off-set value, we get the value of x lying between +1 and -1. The support vectors, guide in classifying the positive value of x, nearest to the value +1 as normal class while negative value of x nearest to the value -1 as attack class. Support Vector Machine(SVM) used in detecting *SQLIA* uses a two dimensional space(X,Y) to find a separating plane for binary classification, in which the error rate is minimal. The SVM uses a portion of the data to train the system, finding several support vectors that represent the training data and guides at runtime to identify a dynamic SQL query into either of the class - attack or normal.

Table 2: Detection approaches and their pros and cons

Name of the method	Approach	Implementation	Pros	Cons
String Evaluation based on initialized trusted list	Query analysis to check initialized unsafe literals	Post Generated	Remove all unsafe literals before execution of runtime query	Developer's dependent and may affect applications functionality
String Evaluation based on initialized untrusted strings	Syntax evaluation for runtime query based on untrusted initialize-tion	Post Generated	Removes tainted strings from runtime query based on initialized trusted strings	Developer's dependent and may affect applications functionality
Parse-Tree Evaluation based on grammar [4]	User originated data is parsed based on predefined grammar	Post Generated	Formulated successfully the structure of a legitimate query	Scope of manipulation of user originated data and may affect applications functionality
Evaluation of Candidate input based on Benign input[5]	The structure of runtime query is analyzed with system generated structure	Post Generated	Elaborated the formulation of SQLIA. A novel approach of guaranteed solution.	Not explored the solution in which it generates mu ltiple runtime stru- cture for one programmer intended query
Proxy based parse tree evaluation [11]	Separate proxy is used in the application site to parse and analyze runtime query	Post Generated	Proxy based data evaluation approach, improves the efficiency of evaluation	Performance also depends on each component of their system architecture and the algorithm for analysis
Program Analysis tool	Taint generation tool	Pre Generated	Identify the vulne- rable mo-dules	Possibility of manipulation
Query language	Retrieval of vulnerabilities	Pre Generated	Reports vulnerable applications	Not always effective

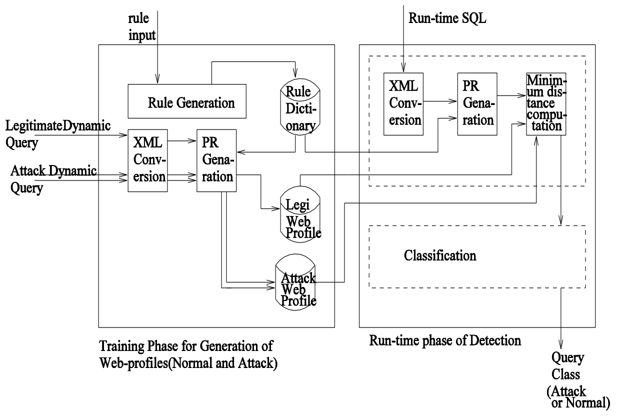

Figure 3: The Architecture of RDUD

For *n* number of queries, the basic input data format and output data domains in SVM are as follows:

$(x_i, y_i), ..., (x_n, y_n)$, $x_i \in R$, $y_i \in \{+1,-1\}$ where, x_i is a two dimensional real inputs vector, R is a set of all real numbers. The goal is to find out the margin of the plane that divides the points having $y_i = 1$ from those having $y_i = -1$. A plane drawn on the set of points $(x_1, x_2, .., x_r)$ satisfying maximum-margin plane and a margin for an SVM trained with samples for a particular class. Samples on the margin are called the support vectors. To identify the class a runtime SQL query is given input to the proposed method of matching process. If it is matched then declared the dynamic SQL query as legitimate(i.e., computed distance ω_1=0). If the *PR* does not matched with any of the *Production Rule* exists in the legitimate web profile then the module generates the unmatched legitimate rules (LR) as output. If the output of execution is the unmatched rules, the algorithm 3 also executed with the attack web profile and input for these modules are - *PR* of the dynamic SQL query and the attack web profile. If it is matched then it declared the dynamic SQL query as attack query(i.e., computed distance ω_2=0). Otherwise, the output of such execution are unmatched attack rules(AR). If the values of both *LR* and *AR* are not empty string then we execute the module called *Distance Computation*. The output of such execution is the number of unmatched rules in *LR* gives the distance (ω_1) with respect to the legitimate *web profile(*WP_1*)*. Then, we execute the module *Distance Computation* by giving input - unmatched rules *AR*. The output of such execution is the number of unmatched rules in *AR*, gives the values of (ω_2) with respect to the attack web profile WP_2. Next, we compute *nearest plane value(x)* for a given value of *b* and classify the runtime SQL query based on the value of *x*. The output of the detection approach *RDUD* may be interpreted with the following cases:

(1) If $\omega_1 = 0$ then it lies in the normal plane and declared as a normal query;

(2) If $\omega_2 = 0$ then it lies in the attack plane and declared as an attack query;

(3) If the value of both ω_1 and ω_2 are non-zero then it computes the value of *x* and plot the
value the two dimensional plane(X,Y), as shown in figure 6 and using trained support vectors it identifies the class. RDUD is significant in view of the following observations:

(a) Initialization of trusted/untrusted strings are avoided;

(b) The matching logic is easy to implement;

(c) Web profile file is adaptively updated;

(d) No restriction is imposed on input strings/ characters;

(e) No restriction of defining syntactical constraint for query structure;

We would like to mention that the structure of the dynamically generated SQL query, may not always be constant depending on the application. To accommodate the variable schema of runtime queries, we convert dynamic SQL queries into semi-structured document (i.e., XML) before generation of web profile. Each rule in *PR* of the runtime SQL query is matched with all the *PR* of the web profile. Thus, the time complexity of such matching of *PR* extracted from *XSQL* and each *PR* of the web profile is $n \times m$; $n \leftarrow$ number of rules available in the XSQL and $m \leftarrow$ number of rules available in a *PR* of the web profile. If T is a set of tags of an XML document, then $\sigma = T \cup \{\$\}$, where "$" stands for the space and carriage return characters. The algorithm generates a context free grammar $G=(V,\sigma,R,S)$; V is a finite set of string tags, R is a finite set of Production Rules and S is the initial string tag of G. A sample generated sequence of *PR* is shown in Table 4. The "Input" column shows the next input string tag, and the "Rules Generated" column shows the production rules. The empty production rule i.e., the start string tag, "$r_0 \rightarrow$" and the next input symbol "< item >" are provided as initial state. In step i and i+1, the non-terminal tagged string, in the form of production rule r_1, r_2, r_3 etc. are generated. The proposed approach is a semi-supervised classification method in which it is not considered the pre-defined parameters to categorize the SQL query strings. It is assumed that the attackers may conduct SQLIA, crafted with malicious strings having variable SQL query structure. The programmer intended SQL query also can have variable structures and are varying with application to application. For a dynamic SQL query the two numeric values - (i) computed distance with respect to the legitimate web profile and (ii) computed distance with respect to the attack web profile are given input to the SVM. The classification at runtime is done based on the plotted distance within the range of values learned during training phase.

4. PROBLEM FORMULATION

A web-application is an application program coded in web-language, executed using web-browser. We define a web-application program as $P = (\{S\},\{F\})$, where $S=\{s_1, s_2, .., s_n\}$ is a set of finite string tags and $F=\{f_1, f_2, .., f_m\}$ is a set of finite web-functions or *url* (static or dynamic). The input vector $(i_1,i_2,...,i_n)$, originated from the user input of web-application P, transports into web-application server and may generate dynamic SQL query *q*. The query *q* then transports into the database server, where executing it and performs DBMS operation, such as - data insertion, data updating, data deletion and retrieving information. The output information converted into web-application program and sends to the client's point, executes by the web-browser. The problem is to form a basis to identify the dynamic SQL query *q* into a 2-class problem(legitimate or attack) with reference to two web profiles - attack web profile and normal web profile.

Definition 3(Rule-Segmented Query): The structure of a dynamic SQL query can be expressed into a set of rules(R), retrieved from the *Rule Dictionary*(RD). Again, each *rule* is expressed by a tuple (rule name, rule type, rule value). *Rule-Segmented Query(RS)* can be defined as a function fRS(q) which returns a set of *rule name* such that each sub-string of *q* has one *rulename*. Thus, we can write,

$f_{RS}(q) : f_{RS}(q) \rightarrow R$; $R = \{r_1, r_2, .., r_n\}$ and $r_i = \{r_{ni}, r_{ti}, r_{vi}\}$.

Let us consider the query string *query*, cited in PHP code given in figure 2, the value of fRS(query) can be written as -

f_{RS}(query)←(K1$A1$K2$T1$K3$Id1$Op1$V1Lop1$Id2 $Lop2$V2)

The function $f_{RS}(q)$ returns the sequence of *rule name* from *RD* corresponding to the sub-strings of the *query*, such that each rule is separated by the literals '$'; A sample shot of *RD* is given in the table 3. In this table it is shown the different columns of *RD* such as - *Rule name*, *Rule type* and *Rule value*.

Definition 4(Rule Dictionary): Rule Dictionary(RD) is a rule data-store, in which each rule is stored as a tuple (Rule Name, Rule Type, Rule Value). Executing a policy or method for generating *RD*, such that it creates an auto-generated unique *rule name*(R_n), which is one of the attribute of *RD* and stores the unique *rule value* into it. Let, the *rule name* stored in *RD* with respect to an web-application, $R_n = (rn_1, rn_2, .., rn_k)$. Corresponding to each *rule name*, r_{ni}, the policy or method retrieves the string from a dynamic SQL query as a *rule value*(r_{vi}) and its *rule type* i.e., Keyword, Attribute etc. Thus, a rule is stored in *RD* in the form of a tuple (r_{ni}, r_{ti}, r_{vi}). A method, f(RD), implements the policy, generates the unique value of *rule name*

Table 3: Rule Dictionary in Tabular form

Rule name [1]	Rule Type [2]	Value [3]
K_1	Keyword	select
K_2	Keyword	from
K_3	Keyword	where
A_1	Attribute	account
A_2	Attribute	users
Id_1	Identifier	username
Id_2	Identifier	password
Rop_1	Relational Operator	=
V_1	Value	ddas
Lop_1	Logical Operator	and
V_1	Value	dd123

and retrieves the values for the other two attributes of *RD* - *rule value*(R_v) and *rule type*(R_t) from a dynamic SQL query. The input to the algorithm of policy or method f(RD) is a dynamic SQL query, q_a and output to the algorithm of policy or method (f_{RD}) retrieves a set of rule tuples. Thus, the data-store of *RD* consists of a set of rows, $\{R_1,R_2, ..,R_p\}$, $p \geq 1$ and we can express a dynamic SQL query, $q_a = R_i = \{(r_{n1}, r_{t1}, r_{v1}), (r_{n2}, r_{t2}, r_{v2}), .., (r_{nk}, r_{tk}, r_{vk})\}$; $R_i \subset$ RD.

Definition 5(Production Rule): A *Production Rule*(PR) is a sequence of *rule name*s, $PR=[r_1,r_2,..,r_n]$, retrieved from *RD* with respect to a sequence of strings available in the structure of a dynamic SQL query. Thus, for each dynamic SQL query(q_a), we have one corresponding production rule(PR_a). $r_i \in N$; $PR_a \subset N$; $N \leftarrow$ set of all *rule names* in *RD*; We consider the example of query string *query* given in the section 2. The value of *PR* extracted from *RD* for *query* can be written as -

$K_1\$A_1\$K_2\$A_2\$K_3\$Id_1\$Rop_1\$V_1\$Lop_1\$Id_2\$Rop_1\$V_2$

Definition 6(web profile): A *web profile* is a summary of dynamic or runtime SQL queries generated for a web-application software. The content of the web profile is a set of finite numbers of unique *Production Rules*(PR), such as $WP=[PR_1, PR_2, .., PR_m]$. According to our approach every web-application software is executed with two web profiles - (i) legitimate web profile(WP_1) and (ii) attack web profile(WP_2). Let us consider the example given in figure 1. The query string *query1* and *query2* are considered as legitimate. However, the query string *a_query* given in section 2 is an attack query. Thus, the legitimate web profile contains the set of *PR*, generated for *query1* and *query2* can be expressed as -

$PR_1=K_1\$A_1\$K_2\$A_2\$K_3\$Id_1\$Rop_1\$V_1\$Lop_1\$Id_3\$Rop_1\$V_3$
$\$Lop_2\$Id_2\$Rop_1\$V_2$
$PR_2=K_1\$A_1\$K_2\$A_2\$K_3\$Id_1\$Rop_1\$V_1\$Lop_1\$Id_3\$Rop_1\$V_4$
$\$Lop_2\$Id_2\$Rop_1\$V_2$
The content of the attack web profile for the query string *a query* can be written as -
$PR_1 = K_1\$A_1\$K_2\$A_2\$K_3\$Id_1\$Rop_1\$V_1\$Lop_1\$Id_3\$Rop_1\$$
$V_3\$ Lop_2\$Id_2\$Rop_1\$V_2\$Lop_2\$Id_4\$Rop_1\$V_4$

Definition 7 (Nearest Plane Value(Pv)): For a given off-set value(b), the plane values(x_1 and x_2) are computed with the values of ω_1 and ω_2 using the equation $(x_1\times\omega_1+b) = 0$ and $(x_2\times\omega_2-b) = 0$. The values of the parameters $\omega(\omega_1$ and $\omega_2)$ and b are constrained to solve the 2-class problem[12]. Thus, the *Nearest Plane value(P*v) can be calculated as $P_v = x_1+x_2$. The constraint parameter *b* called offset parameter and should be chosen such that the range of support vectors, P_v can be projected on the decision surface. Nearest Plane Value(P_v) is the value plotted in (X,Y) plane which gives a measure of the nearest or closest distance from the separating plane. The value of Pv(q) gives a measure of the dynamic query belongs to the class - attack(illegitimate) or normal(legitimate).

Definition 8(Malicious Query): The dynamic SQL query *q* generated with the input data (i_1, i_2, ..., i_n) is a malicious query causing SQL injection attack (SQLIA) if any of the following conditions hold:
• The *PR* generated from the structure of the dynamic query *q* belongs to the attack web profile *WP*2, i.e., $f_{PR}(q) : f_{PR}(q) \leftarrow PR_q \in WP2$;
• The *PR* generated from the structure and the computed *nearest plane value*(Pv) are such that all the following three conditions hold -
(i) $[f_{PR}(q) : f_{PR}(q) \leftarrow PR_q \notin WP_1]$;
(ii) $[f_{PR}(q) : f_{PR}(q) \leftarrow PR_q \notin WP_2]$;
(iii) For a given value of b, the value of P_v is such that $(-1 \le P_v(q) < 0)$.

Definition 9(Legitimate Query): The dynamic SQL query *q* generated with the input data (i_1, i_2, ...,i_n) is a legitimate query if any of the following conditions hold:
• The *PR* generates from the structure of the dynamic SQL query *q* belongs to the legitimate web profile *WP*1, i.e., $f_{PR}(q) \in WP1$;
• The *PR* generated from the structure of the dynamic SQL query q and the computed *value* of P_v are such that it holds all the following three conditions -
(i) $[f_{PR}(q) : f_{PR}(q) \leftarrow PR_q \notin WP_1]$;
(ii) $[f_{PR}(q) : f_{PR}(q) \leftarrow PR_q \notin WP_2]$;
(iii) For a given value of b, the value of P_v is such that $(0 \le P_v(q) \le 1)$.

4.1 Algorithm

In this section we present the steps to implement our approach.

A. Training Phase

Generation of Rule Dictionary(RD): Let us, consider the example *query* string shown in figure 2. The XML equivalent of the query is shown in figure 4. The *Document Type Definition* for the conversion of such query string into XML form is shown in figure 5. The *rule value* <select> is identified by *rule name* - K_1 and *rule type* - *Keyword*. Similarly, the *rule value* <from> is identified by *rule name* K_2 and *rule type Keyword* etc. Thus, using these string values of each dynamic query, the *Rule Dictionary(RD)* is generated. To generate RD, the following steps are executed –

- Read each item - element, attribute, identifier, value, relational operator and logical operator from the XML form of dynamic SQL query(XSQL);
- Check whether the rule value of the corresponding item already exists in the *RD*;
- If the rule value of the corresponding item does not exist in *RD* then execute (i) to (iv) given below:

(i) Generate unique *rule name*(n) for each of the item;
(ii) Retrieve each string of the dynamic SQL query as a *rule value*(v);
(iii) Identify the corresponding *rule type*(t);
(iv) Insert the tuple (n,t,v) into *RD*;

Generation of web profile: To generate web profile from a dynamic SQL query the following steps are to be executed -

- Read dynamic SQL query ;
- Convert the dynamic SQL query into XML form called XSQL;
- Read each item of XSQL - element, attribute and identifiers;
- Retrieve corresponding rule name from *RD* for each item of XSQL;
- Generate Production Rule (PR) of the corresponding dynamic SQL query by constructing the sequence of rule name retrieved in the previous steps;
- Check whether the value of *PR* already exists in the web profile(WP);
- If does not exists, insert *PR* into web profile(WP);

The web profiles generated during training phase will be incrementally updated based on the detection at runtime phase.

Distance Computation: The distance between two SQL queries can be calculated by computing the minimum number of operations required to convert the *PR* of one query into the other. Let, us consider the two *PR* values PR_1 and PR_2 of two different dynamic queries Q_1 and Q_2 as given below :
$PR_1=K_1\$A_1\$K_2\$A_2\$K_3\$Id_1\$Rop_1\$V_1\$Lop_1\$Id_2\$Rop_1\$V_2$
$PR_2=K_1\$A_1\$K_2\$A_2\$K_3\$Id_1\$Rop_1\$V_1\$Lop_1\$Id_2\$Rop_1\$V_2\$Lop_1\$Id_3\$Rop_1\$V_3$
The number of unmatched rule between two *PR*, gives the distance value ω. Thus, the value of ω in the above case between PR_1 and PR_2 becomes 2 due to the unmatched rule - Id_3 and V_3. For a dynamic SQL query the value of ω_1 and ω_2 can be computed at runtime with respect to normal and attack web profiles. If the value of ω_1 is zero, then it declares as ‘Normal’ query. If the value of ω_2 is zero, then it declares as ‘Attack’ query. In case, the values both ω_1 and ω_2 are non-zero, we execute the process for query classification by SVM

```
<select>
<attribute attribute name=balance </attribute>
<from>
<table table name=account </table>
</from>
<where>
<expression>
<identifier identifier name = username </identifier>
<relational operator relational operator == </relational
operator>
<value value = prakash </value> </expression>
<logical operator logical operator = AND </logical
operator>
<expression>
<identifier identifier name=password </identifier>
<relational operator relational operator == </relational
operator>
<value value = p123 </value>
</expression>
</where>
```

Figure 4: XML record based on DTD given in figure 5

```
<!Element Select (attribute)*
<!Element From (table)*
<!Element Where (expression, (Logical Operator,
Expression)*)
<!Element Expression(Identifier, Relational Operator, Value)
<!Element Logical Operator(AND|OR|NOT)
<!Element Identifier(#pcdata)
<!Element Relational Operator (= | != | < | > | <= | >= |)
<!Element Value (#cdata)
```

Figure 5: Document Type Definition (DTD) of XML equivalent of SQL query

During training phase, we perform tests with dynamic SQL queries (both legitimate and attack). For each matching process we compute the value of x_1 and x_2 and finally, the *nearest plane value* NPV(x) = x_1+x_2. We experimented with both normal and attack queries and plotted the value of *x* in a two dimensional plane(X,Y). A separating plane which separates both the dimensional planes(X and Y) equally is drawn, shown in figure 6. We also draw *support vector plane* by joining the plotted points, parallel to the separating plane. The plotted points below the separating plane and towards the X-axis are identified as normal class, and the plotted points above the separating plane towards the Y-axis are identified as attack class. The following steps are executed for the computation of distance value(ω) between *PR* of two queries.

• For all the *m* numbers of *PR* of a web profile(WP_1) do {
• For all the rules of *PR* for dynamic runtime SQL query do {
• Calculate the number of unmatched rules to each *PR* of WP_1, $\{C_1,C_2,C_3, ..C_m\}$ } }
• Retrieve the minimum value C_k from $\{C_1,C_2, ..,C_m\}$ which is the distance between the runtime SQL query with respect to the web profile WP_1

Thus, the distance value may be computed at runtime between the *PR* value of dynamic SQL query with each *PR* value of both attack and normal web profile separately.

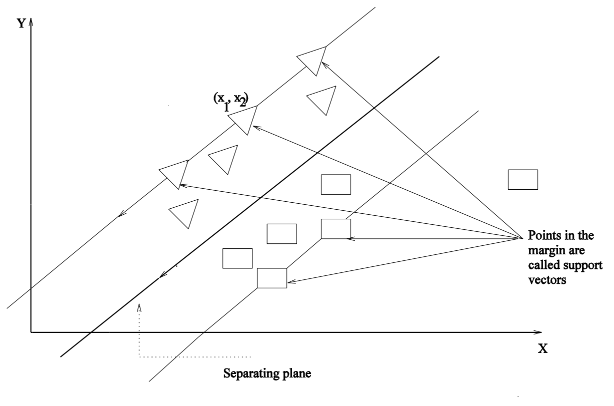

Figure 6: The plane of SVM

B. Runtime Phase(Query Classification):

To convert *XSQL*(XML form of runtime query) into sequence of rules called *PR*, we use the algorithm based on the SEQUITUR algorithm proposed by Nevill-Manning [14]. The algorithm for *PR* generation from a runtime SQL query is given in the algorithm 1. The algorithms for distance computation and the final classification based on *PR* matching are given in algorithm 2 and 3 respectively. A sample of the steps in generating *PR* is given in the table 5.

Algorithm 1: PR Generation

```
Data: XML document, Rule Dictionary(RD)
Result: Production Rule PR
begin
    for all item in the XML document do
        Replace one or more item as rule r based on Rule Dictionary ;
        Append to the end of production rule PR ;
    end
    if item is not available in Rule Dictionary then
        Replace with a new rule ;
        Update Rule Dictionary with new rule ;
    end
    if duplicate rule appears then
        Delete Duplicate rule ;
    end
    if every rule r in PR is used only once then
        Declare ('Production Rule(PR) generation is complete') ;
        Return PR ;
    end
    else
        Repeat step [e] ;
    end
end
```

Algorithm 2: Distance Computation between PR of two Queries

```
Data: minimum_unmatched_rule, WP_PR[], Minimum_count_u, m;
Result: Rule_edit_distance;
begin
    for (i = 1 to Minimum_count_u) do
        Read minimum_unmatched_rule[i];
        for (j=1 to m) do
            r ← rule_count(WP_PR[j]);
            for (k = 1 to r) do
                c = character_count(minimum_unmatched_rule[i]);
                for (l=1 to c) do
                    if (minumin_unmatched_rule[] ≠ WP_PR[l][k]) then
                        pr_edit_distance[k] ← pr_edit_distance[k] + 1;
                    end
                end
            end
            wp_edit_distance[j] ← Minimum(pr_edit_distance[];
        end
    end
    rule_edit_distance = Minimum(wp_edit_distance[]);
end
```

Algorithm 3: PR Matching of two dynamic SQL queries

```
Data: web-profile, PR of dynamic query
Result: minmum_unmatched_rules, m, Minimum_count_u
begin
    unmatched_rules[] ← null;
    minimum_unmatched_rules[] ← null;
    tag ← 0;
    distance ← 0;
    m ← number of PR in the web-profile;
    for (i=1 to m) do
        if (PR_Runtime = WP_PR[i]) then
            tag ← 1;
        end
    end
    if (tag = 1) then
        minimum_unmatched_rule ← 0;
    end
    else
        for (i=1 to m) do
            n= count_rule[PR_Runtime];
            for (j=1 to n) do
                if (PR_Runtime[j] ≠ WP_PR[i][j]) then
                    unmatched_rules[i] ← strcat(unmatched_rules, PR_Runtime[j]);
                end
            end
        end
    end
    Count_u[] ← 0;
    minimum_unmatched_rules ← 0;
    for (t=1 to m) do
        Count_u[t] ← character_count(unmatched_rules);
    end
    Minimum_Count_u = MIN(Count_u[]);
    for (t = 1 to m) do
        if (character_count(unmatched_rules[t] = Minimum_Count) then
            (minimum_unmatched_rules ← unmatched_rules[]);
        end
    end
end
```

Table 4: Steps in generating Production Rules(PR)

Step	Input	Rules Generated
1	<item>	$r_0 \rightarrow$
2	\$	$r_0 \rightarrow$<select>
3	<item>	$r_0 \rightarrow$<select>\$
4	\$	$r_0 \rightarrow$<select>\$<attribute>
5	<item>	$r_0 \rightarrow$<select>\$<attribute>\$
6	\$	$r_0 \rightarrow$<select>\$<attribute>\$<name=balance>
7 ….	<item>	$r_0 \rightarrow$<select>\$<attribute> \$<name=balance>\$
i	<item>	$r_0 \rightarrow$<select>\$<attribute>\$name=balance\$ >\$</attribute>\$<from>…</from>\$
i+1	\$	<select>$r_1 r_2$ … </from>\$ r_0\$</attribute>r_3 … </where> $r_1 \rightarrow$\$<attribute> $r_2 \rightarrow$\$<name=balance>

5. EXPERIMENTAL RESULTS

In our experimental setup the detection modules of *RDUD* are installed in a web-application server of a three-tire application architecture. The web-application server is connected with a database server and is accessible to web clients. We generate two master files for each application. The first file consists of the legitimate dynamic SQL queries generated with legitimate access during training phase. The other file consists of the attack queries generated, corresponding to each intended structure of SQL queries defined in web-application program.

Using the master file having legitimate SQL queries we generate the legitimate web profile(WP_1). Similarly, we generate attack web profile(WP_2) with master file having the attack SQL queries. Both the profiles WP_1 and WP_2 are stored in the web-application server. The server also contains the modules for - (i) *Rule Dictionary* generation; (ii) *Production Rules* generation; (iii) *Distance Computation* and (iv) *Query Classification*. We used MySQL in the database server. To test the detection rate using our approach, we simulate the runtime environment using two different files - *file 1* and *file 2*. The *file 1* consistS of legitimate dynamic SQL queries and *file 2* consistS of attack or illegitimate dynamic SQL queries. While generating the *file 1* and *file 2* we include dynamic SQL queries in it such that the corresponding *PR* of some of the queries exist in the respective web profiles, while other does not exists. To generate the web profiles, at first we construct the *Rule Dictionary* table and executed the module for generation of *Production Rule(PR)* using algorithm 1, from both the master files - legitimate master file and attack master file. Then the values of *PR* are stored in WP1 and WP2 respectively. The web-application is considered for our experiment, consists of three programmer intended SQL queries. The web profile(WP_1) consists of 50 numbers of *PR* values of dynamic SQL queries, which are considered as legitimate. Similarly, the web profile(WP_2) consists of 60 numbers of *PR* values of attack SQL queries such that for each class of SQL injection attack, we considered 12 numbers of attack SQL queries. While computing the distance value (ω_1 and ω_2), if we avoid the value portion of an unmatched expression, we do not get 100% detection rate for some of the attack classes of SQLIA such as - ByAut, IUnion, Adqry, UexRP. This is due to the fact that the structure of some of the attack queries available in the attack web profile may be same with the structure of the legitimate queries. According to our concept, depending upon application requirement, the structure of the attack SQL queries may be same with the structure of the application generated variable dynamic query. We finally experimented with the runtime query structure which includes the value portion. We changed the *Distance* value computed for an unmatched value in an expression of a runtime SQL query. Thus, for each unmatched value we considered the *Distance* value as 1. In this experiment we got 100% detection rate We also experimented by considering the character wise edit distance method instead of considering the *Distance* value for unmatched *rule value* as 1. In this case also we got 100% detection rate for all the attack classes except the attack class *ByAut*. This is due to the fact that any new dynamic SQL queries consist of new value, can increase the distance. Thus, according to our experimental analysis, we should apply our detection approach including the character wise distance computation for all the attack classes of SQLIA except the attack class *ByAut*. For the detection of this particular attack class *ByAut* we should apply structure matching excluding the value of a dynamic SQL query. We tabulated the value of x_1, x_2 and the corresponding value of x in table 5. The accuracy of the detection of query class is also cited in the table 5.

6. CONCLUSIONS AND FUTURE WORK

Our detection method RDUD has been found to perform satisfactorily over a test data-set. The method is also found capable of handling the variable structure of dynamically generated SQL query based on user input. The effectiveness of updating mechanism of the web profile can be increased with additional checking module. However, special care needs to be taken for maintaining the integrity of the web profile files to avoid poisoning of web profiles. Also, the use of appropriate encoding technique can help to avoid stolen key attack.

Table 5: Values of x_1, x_2, x and query class detection

d_1	d_2	b	x_1	x_2	x	Query Class	Correct/Wrong
0	-	-	-	-	-	Normal	Correct
	0	-	-	-	-	Attack	Correct
	0	-	-	-	-	Attack	Correct
0	-	-	-	-	-	Normal	Correct
2	7	0.5	0.25	-0.07	0.32	Normal	Correct
1	7	0.5	0.5	-0.07	0.43	Normal	Correct
6	2	0.5	0.08	-0.25	-0.17	Attack	Correct
7	1	0.5	0.07	-0.5	-0.43	Attack	Correct

7. REFERENCES

[1] Debasish das, Utpal Sharma & D.K. Bhattacharyya. An approach to detection of SQL injection attack based on dynamic query matching. International Journal of Computer Applications, 1(25), 2010.

[2] Common Weakness Enumeration. 2011 cwe/sans top 25 most dangerous software errors. MITRE Corporation, http://cwe.mitre.org/top25/#Listing, 2011.

[3] SPAM fighter products 2003-2011, CISCO Worldwide. A growing menace. http://spamfighter.com/News-15078-Sql-Injection-Attacks-A-Growing-Menace.htm, 2010. reports Forman, G. 2003. An extensive empirical study.

[4] Reports from Information System and Audit Cell(Indian Bank), Chennai(INDIA). Audit report for Core banking/net banking/ mobile banking/ atm/ data center/ dr site/ networking infrastructure and other integrated systems.

[5] Z. Su and G. Wassermann. The essence of command injection attacks in web application. In the 33rd Annual Symposium on Principlas of Programming Languages, pages 372-382, January 2006.

[6] P. Madhusudan, Prithvi Bisht and V.N. Venkatakrishnan. Dynamic candidate evaluation for automatic prevention of SQL injection attacks. ACM Transactions on Information and System Security, 13, 2(14), 2010.

[7] C. Anley. Advanced SQL injection in SQL server applications. White Paper, Next Generation Security Software, http:/wwwgenss.com/papers/advanced sql injection.pdf, 2002.

[8] D. Litchfield, Director of Security Architecture. Web application dissembly with odbc error message, a report. http://www.atstake.com.

[9] M. Howard and D Le Blane. Writing Secure Code, volume II. Microsoft Press, Redmond, Washington, 2003.

[10] Jeremy Viegas William G.J. Halfond and Alessandro Orso. A classification of SQL injection attacks and countermeasures. In Proceedings of the IEEE International Symposium on Secure Software Engineering, Arlington, VA, USA, 2006.

[11] Yi Yuan Anyi Liu and Duminda Wijesekera. Sqlprob: a proxy based architecture towards preventing sql injection attacks. ACM Digital Library, 2009.

[12] Vladimir Vapnik Cornna Cortes. Machine Learning, 20, 273-297(1995).

[13] K. Krithivasan and R. Sitalakshmi. Efficient two dimensional pattern matching in presence of errors. Information Sciences, 43, 1987.

[14] James Law. Path based dynamic impact analysis. In IEEE Explore, Computer Science Department, Oregon State University, 2003.

[15] Steve R. Gunn. Support Vector machines for classification and regression. Technical report, Faculty of engineering, science and mathematics. School of Electronics and Computer Science. ecs.soton.ac.uk/srg/publications/pdf/SVM.pdf, 1998.

[16] F. Bouma. Stored Procedures are Bad, O'Kay, Technical Report. Asp.Net/Weblogs, http://weblogs.asp.net/fbouma /archieve/2003/11/18/38178.aspx, November 2003.

[17] E.M. Fayo. Advanced sql injection databases, technical report. Agencies Information Security, Black hat Briefings, Black Hat U.S.A, 2005.

[18] S. McDoland. SQL Injection, modes of attack, defence and why it matters. GovernmentSecurity.org, April 2006.

[19] F. Finigan. SQL injection and Oracle – part 1 and part 2. Security Focus, November 2002.