Blocking SQL injection in Database Stored Procedures

Thesis submitted in partial fulfillment

of the requirements for the degree of

Bachelor of Technology

In

Computer science and engineering

By

Shrinivas Anand Panchamukhi & Abhilash Sarangi

(Roll numbers: 10606034 & 10606047)

Department of computer science and engineering

National Institute of Technology Rourkela

Rourkela Orissa India

Blocking SQL injection in Database Stored Procedures

Thesis submitted in partial fulfillment

of the requirements for the degree of

Bachelor of Technology

In

Computer science and engineering

By

Shrinivas Anand Panchamukhi & Abhilash Sarangi

(Roll numbers: 10606034 & 10606047)

Under the guidance of

Prof. K.S Babu

Department of computer science and engineering

National Institute of Technology Rourkela

Rourkela Orissa India

National Institute Of Technology Rourkela
Rourkela Orissa

Certificate

This is to certify that the work in this thesis report titled "**Blocking SQL injection in Database Stored Procedures**" by **Shrinivas Anand Panchamukhi & Abhilash Sarangi** has been carried out under my supervision in partial fulfillment of the requirements for the degree of **Bachelor in Technology** in Computer Science and Engineering during the session 2009-2010 in the department of Computer Science and Engineering, National Institute of Technology Rourkela and his work has not been submitted else where for a degree.

Place: Rourkela

(Prof. K.S Babu)
Department of CSE

Date: April 25, 2010

ACKNOWLEDGEMENTS:

Many people have been involved, directly or indirectly, in the completion of this thesis and we would like to take this opportunity to express our gratitude to them. We would firstly like to sincerely thank our project guide, Prof K. S. Babu, for being a constant source of inspiration throughout the course of this project work and this work would not have been possible without his valuable guidance. We would like to express our gratitude to all the research scholars whose works were referred to by us during the completion of this project work. We would also like to take this opportunity to thank all of our branch mates for their encouragement and guidance at all times during the course of this project work. A special mention about Serge Gorbenko for his SQL parser tool, which has been used in the implementation of the Runtime Analyzer.

Shrinivas Panchamukhi (10606034)

Abhilash Sarangi (10606047)

ABSTRACT:

This thesis contains a summary of all the work that has been done by us for the B-Tech project in the academic session of 2009-2010. The area chosen for the project was SQL Injection attacks and methods to prevent them, and this thesis goes on to describe four proposed models to block SQL Injection, all of them obtained from published research papers. It then gives the details of the implementation of the model "*SQL Injection prevention in database stored procedures*" as proposed by K. Muthuprasanna et al, which describes a technique to prevent injections attacks occurring due to dynamic SQL statements in database stored procedures, which are often used in e-commerce applications. The thesis also contains the algorithms used, data flow diagrams for the system, user interface samples and the performance reports. The particulars of some of the modifications made to the proposed model during implementation have also been documented, and there has also been included a section which discusses the possible updations that could be made to the tool, and future work.

Contents

1.0 Introduction 10
1.1 Motivations 10
1.2 Research focus and original contributions 10
1.3 Structure of the work 11
2.0 SQL Injection 12
2.1 SQL Injection Attacks 12
2.2 Types Of Attacks 12
3.0 Avoiding SQL Injection 15
3.1 Parameterized Queries with Bound Parameters 15
3.2 Parameterized Stored Procedures 15
4.0 Existing Prevention/Detection Models 17
4.1 Secure SQL Processing 17
4.1.1 Proposed architecture 19
4.1.2 Performance 20
4.1.3 Shortcomings of the technique 20
4.2 Weight-based Symptom Correlation Approach 21
4.2.1 Detection Approach 23
4.2.2 Performance 24
4.3 Preventing SQL Injection in Stored Procedures 25
4.3.1 Basics 25
4.3.2 SQL Injection in stored procedures 25
4.3.3 Proposed Solution 27
4.4 MUSIC: Mutation Based SQL Injection Checking 30
4.4.1 Mutation operators 31
4.4.2 Mutant Killing Criteria 32
4.4.3 Details of Mutation Operator 33
5.0 Key Observations 35
6.0 Implementation Details 37
6.1 DFD's for Static Analyzer 37

6.1.1 Level 0 37
6.1.2 Level 1 37
6.1.3 Level 2 38
6.2 DFD's for Runtime Analyzer 39
6.2.1 Level 0 39
6.2.2 Level 1 40
6.3 Graphical user interface 41
6.4 Structure Description and Technical Specifications 43
6.5 Pseudo code 44
6.5.1 Static Analyzer 44
6.5.2 Runtime Analyzer 45
7.0 Results and Performance 46
7.1 Results 46
7.2 Performance analysis 47
7.2.1 Static Analysis 47
7.2.2 Runtime Analysis 50
8.0 Conclusions and Future Work 55
9.0 References 56

List of figures

Figure number	Title of figure	Page number
1	Node structure of the main doubly linked list	13
2	Node structure of singly linked list	14
3	Correlation Process for weight-based approach	17
4	Code sample for stored procedure vulnerable to SQLIA	22
5	SQL-Graph representation	24
6	SQLIA Detection : SQL-FSM Violation	25
7	The proposed Operators in MUSIC	27
8	Mutant killing criteria	28
9	Table ‘tlogin’	29
10	Example Applications of RMWH	29
11	Example Applications of NEGC	30
12	Level 0 DFD for Static Analyzer	33
13	Level 1 DFD for Static Analyzer	33
14	Level 2 DFD for Static Analyzer	34
15	Level 0 DFD for Runtime Analyzer	35
16	Level 1 DFD for Runtime Analyzer	36
17	Main working window	37
18	View stored procedure text	38
19	Graph: No. of queries vs execution time for SA	44
20	Graph: No. of dependencies vs execution time for SA	45
21	Graph: No. of dependencies vs execution time for RA	47
22	Graph: No. of queries with one depedency vs execution time for RA	48
23	Graph: no. of queries with 2 dependencies vs Execution time for RA	49
24	Graph no. of queries with three dependencies vs Execution time for RA	50

List of Tables

Table Number	Title of table	Page number
1	Summary of testing results	43
2	Execution time for different no. of queries for SA	44
3	Execution time for different no. of dependencies for SA	45
4	Execution time for different no. of dependencies for RA	46
5	Execution time for queries with 1 dependency each	48
6	Execution time for queries with 2 dependencies each	49
7	Execution time for queries with 3 dependencies each	50

1.0 Introduction

1.1 Motivations

Information can be said to be the single most important business asset today and achieving a high level of information security can be viewed as imperative in order to maintain a competitive edge. SQL Injection Attacks (SQLIA's) are one of the most severe threats to web application security. They are frequently employed by malicious users for a variety of reasons like financial fraud, theft of confidential data, website defacement, sabotage, etc. The number of SQLIA's reported in the past few years has been showing a steadily increasing trend and so is the scale of the attacks. It is, therefore, of paramount importance to prevent such types of attacks, and SQLIA prevention has become one of the most active topics of research in the industry and academia. There has been significant progress in the field and a number of models have been proposed and developed to counter SQLIA's, but none have been able to guarantee an absolute level of security in web applications, mainly due to the diversity and scope of SQLIA's. One common programming practice in today's times to avoid SQLIA's is to use database stored procedures instead of direct SQL statements to interact with underlying databases in a web application, since these are known to use parameterised queries and hence are not prone to the basic types of SLQIA's. However, there are vulnerabilities in this scheme too, most notably when dynamic SQL statements are used in the stored procedures, to fetch the database objects during runtime. Our work is centred on this particular type of vulnerability in stored procedures and we develop a scheme for detection of SQLIA in scenarios where dynamic SQL statements are used.

1.2 Research focus and original contributions

Our project work is focused on detection of SQLIA's in database stored procedures which contain dynamic SQL statements. After carefully analyzing the mode of the attack in such a scenario, and examining previously proposed models, we decided to implement the model proposed by Wei, Muthuprasanna, et al, which was merely a statement of the model to be used. We are wholly responsible for development of the algorithm for the implementation of the idea and all the programming concepts and specifications involved. Our algorithm essentially consists of two components, one being the static analyser which is an application to be used by either the DBA or the developer, and the other being the runtime analyser which is a library for use by the developer. The model works by comparing the structures of a query before and after including the parameters contained in it. If there is a

conflict in the structures (which would be the case in the event of an SQLIA) then the query is flagged as an SQLIA and reported back to the application for further action.

1.3 Structure of the work

The rest of this work is organized as follows. In Chapter 2, the basics of SQLIA are introduced and different types of attacks are discussed, with illustrations of their structure and mode of attack. Chapter 3 contains a description of the basic techniques used to prevent SQLIA's in web applications. These include the usage of parameterised queries, parameterised stored procedures, and least privilege connections, which are discussed in some detail. In Chapter 4, the various SQLIA detection and prevention models studied by us have been documented with a detailed description of their working and specific advantages and disadvantages of each. Chapter 5 contains the key observations from the models presented in Chapter 4, and then goes on to identify which model would be most suitable for implementation for the purpose of this thesis work. Chapter 6 contains documentation of the programming details for the software to be made, including the Data Flow Diagrams for both the static and the runtime analyzers, a few samples of the graphical user interface of the software, the structure description and technical specifications of the software, and the pseudo-code for both the static and the runtime analyzers. Chapter 7 reports the performance parameters of the software and the results, and Chapter 8 talks about the possible updation and future work that could be done to improve the functionality and efficiency of the developed software.

2.0 SQL Injection

2.1 SQL Injection Attacks

SQL injection vulnerabilities have been described as one of the most serious threats for Web applications. Web applications that are vulnerable to SQL injection may allow an attacker to gain complete access to their underlying databases. Because these databases often contain sensitive consumer or user information, the resulting security violations can include identity theft, loss of confidential information, and fraud. In some cases, attackers can even use an SQL injection vulnerability to take control of and corrupt the system that hosts the Web application. The major cause of SQL injection attacks is inefficient user input validation and poor programming practices. However, there are advanced injection techniques which exploit the inherent shortcomings of programming languages and the underlying databases.

The typical intentions of the attacker performing a SQL injection attacks may be to:

- Identify inject-able parameters.
- Perform database finger-printing.
- Determine the database schema.
- Extract and modify data.
- Perform Denial of Service (DoS)
- Bypass authentication and perform privilege escalation
- Execute remote commands

2.2 Types Of Attacks

The basic types of attacks are as follows:

- **Tautology attacks**: The basic objective of a tautology-based attack is to inject code into one or more conditional statements so that they always evaluate to true. The most common usages are found to be in bypassing authentication pages and in extracting data. In this type of injection, the attacker exploits an inject-able field contained in the WHERE clause of a query. He transforms this conditional into a tautology and hence causes all the rows in the database table targeted by the query to be returned. Typically, the injection attempt is said to be successful when the code either displays all the returned records or performs some action if at least one record has been returned.
  Eg : ***SELECT accounts FROM users WHERE***
  login='' or 1=1 -- AND pass='' AND pin=

The code injected in the conditional (OR 1=1) transforms the entire WHERE clause into a tautology. The returned set evaluates to a non null value, which causes the application to conclude that the user authentication was successful.

- **UNION Attacks***:* Here, an attacker exploits a vulnerable parameter to alter the data set returned by a given query. Using this technique, an attacker can trick the application into returning data from a table different from the one that was intended by the developer. Attackers do this by injecting a statement of the form:
  UNION SELECT <rest of injected query>. Because the attacker is in complete control of the second/injected query, he can use that query to retrieve information from any desired table in the database. The result of this attack is that the database returns a dataset that is the union of the results of the original/first query and the results of the injected/second query.

  Eg : ***SELECT accounts FROM users WHERE login='' UNION SELECT cardNo from CreditCards where acctNo=10032 -- AND pass='' AND pin=***

  Assuming that there is no login equal to "", the original/first query returns the null set, whereas the second query returns data from the "CreditCards" table. In this case, the database would return column "cardNo" for the account "10032."

- **Logically incorrect query attacks**: This type of attack lets an attacker gather important information about the type and structure of the back-end database in a Web application. The attack is considered to be a preliminary, information gathering step for subsequent attacks. The vulnerability leveraged by this type of attack is that the default error page returned by application servers is often overly descriptive, which can serve to expose sensitive information about the databases to the hacker. In fact, the simple fact that an error message is generated can often reveal vulnerable/inject-able parameters to an attacker.

  Eg **:** ***SELECT accounts FROM users WHERE login='' AND pass='' AND pin= convert (int,(select top 1 name from sysobjects where xtype='u'))***

  In this attack string, the injected SELECT query attempts to extract the first user table (xtype='u'). The query then tries to convert this table name into an integer. Because this is not a legal type conversion, the database throws an error. For Microsoft SQL Server, the error would be:"*Microsoft OLE DB Provider for SQL Server (0x80040E07) Error converting nvarchar value 'CreditCards' to a column of data type int."* There are two useful pieces of information in this message that aid an attacker. First, the attacker can see that the database is an SQL Server database, as the error message explicitly states this fact. Second, the error message reveals the value of the string that caused the type conversion to occur.

- **Piggybacked Query** : In this attack type, an attacker tries to inject additional queries along with the original query, which are said to "piggy-back" onto the original query. As a result, the database receives multiple SQL queries for execution. The first is the intended query which is executed as normal; the subsequent ones are the injected queries, which are executed in addition to the first. This type of attack can be extremely harmful. If successful, attackers can insert and execute virtually any type of SQL command, including stored procedures, into the additional queries and have them executed along with the original query. Vulnerability to this type of attack is often dependent on having a database configuration that allows multiple statements to be contained in a single string.

  Eg : ***SELECT accounts FROM users WHERE login='doe' AND pass=''; drop table users -- ' AND pin=123***

  After completing the first query, the database would recognize the query delimiter (";") (which is used to execute multiple queries in succession) and execute the injected second query. The result of executing the second query would be to drop the table 'users', which would likely destroy valuable information. Many databases do not require a special character to separate distinct queries, so simply scanning for a query separator is not an effective way to prevent this type of attack.

  Apart from these basic types, there are also other types of injection attacks like inference-based queries, timing attacks and alternate encodings, which are outside the scope of this thesis work.

3.0 Avoiding SQL Injection

There are two complementary yet greatly successful methods[5] of mitigating SQL Injection attacks:

- Parameterized queries using bound, typed parameters
- Careful use of parameterized stored procedures.

Parameterized queries are the easiest to adopt, and work in fairly similar ways among most web technologies in use today, including:

- Java EE
- .NET
- PHP

3.1 Parameterized Queries with Bound Parameters

Parameterized queries[5] keep the query and the data separate through the use of placeholders known as "bound" parameters. This helps in preventing SQLIA by not allowing the structure of the query to be altered; rather it merely “fills in” the input parameters into their positions and keeps the rest of the query structure intact. Since a majority of the SQLIA techniques rely on altering the query structure for injection attacks, this serves as a very effective combative technique. For example, in Java, this looks like:

"select * from table where columna=? and columnb=?"

The developer must then set values for the two ? placeholders. Note that using this syntax without actually using the placeholders and setting values provides no protection against SQL injection.

3.2 Parameterized Stored Procedures

The use of parameterized stored procedures[5] is an effective mechanism to avoid most forms of SQL Injection. If used in combination with parameterized bound queries, it makes it very unlikely for SQLIA’s to occur within an application. However, the use of dynamic code execution features can allow SQL Injection as shown below:

create proc VulnerableDynamicSQL(@userName nvarchar(25)) as

```
declare @sql nvarchar(255)

set @sql = 'select * from users where UserName =  + @userName + '

exec sp_executesql @sql
```

In the example shown above, it can be seen that the comparison element is being appended to the query at runtime, and hence this type of dynamic code is vulnerable to SQLIA's. Typically, dynamic code is used when database object names are to be passed at runtime.

3.3 Least privilege connections

Another effective way of avoiding SQLIA's is by ensuring that any application that has access to the underlying databases should only use accounts which grant it the minimum permissions necessary to access the objects that it needs to use[5]. Under no circumstance should any such application be allowed to use accounts such as "dba" or "admin", which grant it full privileges to alter the database and extract data in any which way it can.

4.0 Existing Prevention/Detection Models

4.1 Secure SQL Processing

This model has been proposed by Dibyendu Aich, an M-tech research scholar at the National Institute of Rourkela in the research paper titled "*Secure Query processing by blocking sql injection"*[3]. The basic mechanism that this model uses is a two phase query analysis, consisting of the static analysis phase and the dynamic analysis phase. During runtime, the model checks the input query structure with the previously stored query structure to determine possible SQLIA's. The database of the valid query structures is made statically, during compilation. The valid structures are stored as a singly linked list of the different tokens in a sequential ordering. All such valid query structures in the application are then stored as a doubly linked list where each node of the doubly linked list contains the staring address of an individual singly linked list of a valid query structure. So basically, when a new query is sent to the database server, the model starts searching for a match of the structure of the query in the linked representation. If a match is found, the search is stopped and the query is dubbed a valid query, else it is labelled as an SQL injection attack.

In effect, the searching procedure is the operation of checking if the sequence of query language tokens generated by the arrived query is the same as the sequence of tokens generated by at least one singly linked list of valid query structures, upon finding which, the query is sent to the server for execution. This model makes use of the SQL parser of the backend database to parse the incoming query into a sequence of tokens, with an additional field to denote if the node is a user input or if it is a token of the static part of the query. When a node denoting a user input is found in the linked list, the checker skips right past it to the next static token, and the matching continues.

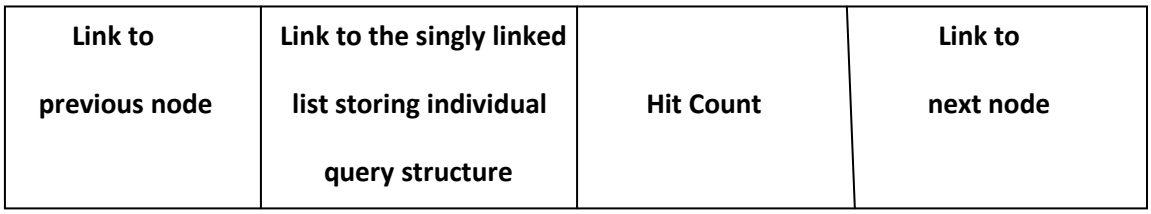


Figure 1: Node structure of the main doubly linked list[3]

Data related to the tokens of a valid query	Is it a user input?	Link to next node

Figure 2: Node structure of singly linked list for storing a valid individual query structure[3]

From the above description of the matching technique, it is clear that for a successful search, number of tokens in the input query must be equal to the length of the linked list storing its structure.

Key advantages over other token matching algorithms :

- Although this process is a relatively secure way of checking for SQL injection, it is computationally very intense because it involves searching of the linked list for a matching structure. For a database where the number of valid query structures is very huge, this could take an unacceptably long time. Hence, the technique proposed to deal with this shortcoming is to use a **multithreaded search**, where the input query is checked with each different query structure running as different threads. When a thread performs a successful search, it intimates all other running threads to immediately terminate.

- However, due to hardware constraints, there is always a limit on the number of simultaneously executing threads in any application. Hence, to counter this, a technique is used where we check for the matching structure with stored structures based on a priority search, where the queries that are used more regularly are given a higher priority compared to those used rarely, which can be easily maintained by associating a **hit counter** with each query structure which is incremented each time the particular structure is matched with an incoming query.

- For runtime matching, if a conventional literal matching is used to compare tokens, it will lead to a huge computational complexity. For example, if there are ‘n’ literals in the incoming query and ‘q’ individual valid query structures of the

same length as the input query, the worst case complexity will be O(n*q). To avoid this overhead, a technique is used where instead of using literal string matching algorithms; each token is simply mapped to **an integer value**. These integer values are also stored in the database instead of the literals as the query fingerprint. When an input query arrives for checking, each token of that query is replaced by its corresponding integer value, and then performing direct integer comparison checks instead of token matching, thus greatly reducing the computational overhead and also significantly reducing the memory space required.

The formula used to convert a token into its corresponding integer value is to multiply each ASCII decimal value of a literal by its position number in the token, and then sum it up. For example, let's consider the keyword, 'SELECT', the corresponding ASCII decimal values are S=83, E=69, L=76, E=69, c=67, T=84; and the position of each literal is S=1, E=2, L=3, E=4, C=5, T=6. So, after multiplying the ASCII values of each literal with its position and summing them up, we get, 83*1 + 69*2 + 76*3 + 69*4 + 67*5 + 84*6 = 1564. Hence, the corresponding integer value of 'SELECT' is 1564. The integer equivalents of all other keywords can be similarly evaluated.

- It is known that for a valid incoming query, the number of tokens is the same as the number of tokens in its corresponding query structure in the database. Therefore, to reduce the search space, all the valid structures having the **same length are grouped together**. For an input query, first its length is calculated, and it is only compared with that group of valid structures which have the same length. To achieve this, an array is used each element of which contains the starting address of a doubly linked list which again contains the starting addresses of all the singly linked lists of the valid structures of a particular length. Hence, this array contains one element for each different possible query length in the application, with the cell number indicating the query length. This substantially reduces the search space and optimizes the searching process.

4.1.1 Proposed architecture

This scheme would be implemented as a different layer in between the application and the database. It would perform as a virtual database to the application, as it would take the queries from the application program, analyse them, and if found safe, then send

them to the database and subsequently send the result set back to the application. As this scheme is totally dependent of the token generation, for which it uses the DBMS parser, it would be specific to different databases as different databases use different keyword sets and function names, as well as different syntax. Hence, we see that it is a wise choice to use the database parser to perform the parsing.

4.1.2 Performance

The main advantage of this model is that since it is multithreaded in nature, it can utilize the features of the modern multi-core processors very efficiently. The basic complexity of this algorithm is in three procedures:

1. **Token separation** : This depends entirely on the database involved because it is wholly dependent on the database parser, since most databases have a different keyword set, syntax and function names. Thus, this factor can be taken to be the same for all implementations.
2. **Token to integer conversion :** This is of the order O(n) where 'n' is the total number of unique literals in all the queries put together.
3. **Searching :** Worst case is when it is an unsuccessful search, or when the match is found in the last linked list of any group. If the length of the singly linked list is 'm' and there are 'q' such linked lists, then the search complexity is O(m*q). The best case complexity will be if we found a match in the first structure, in which case the order will be O(m). If we had used a literal wise checking, then the complexity would have been O(n*q), where 'n' is the total number of literals in the query and n>>m.

4.1.3 Shortcomings of the technique

- It can only detect injection attacks where the structure of the query is changed.
- This model can only process a standalone SQL query, but does not work for PL/SQL code block.

4.2 Weight-based Symptom Correlation Approach

This technique has been proposed by Massimo Ficco et al in the research paper titled "*A Weight-Based Symptom Correlation Approach to SQL Injection Attacks"*[4]. In this technique, a number of symptoms of an SQL injection attack are considered which appear in different times, involve different components and produce several alert events. Correlating these symptoms, which are diverse in nature and detected by distributed probes, allows us "to build a unified view of the web service security, as well as simplifies the recognition of intrusive behaviours".[4]

Correlation process : Correlation is a process that receives as input detected symptoms from many distributed probes. During this process, symptoms are analyzed and merged into compact reports, which describe the security status of the monitored web applications, which is followed by a confidence assessment of the produced reports. As shown in the figure below[4], the steps performed by the considered correlation process are the following:

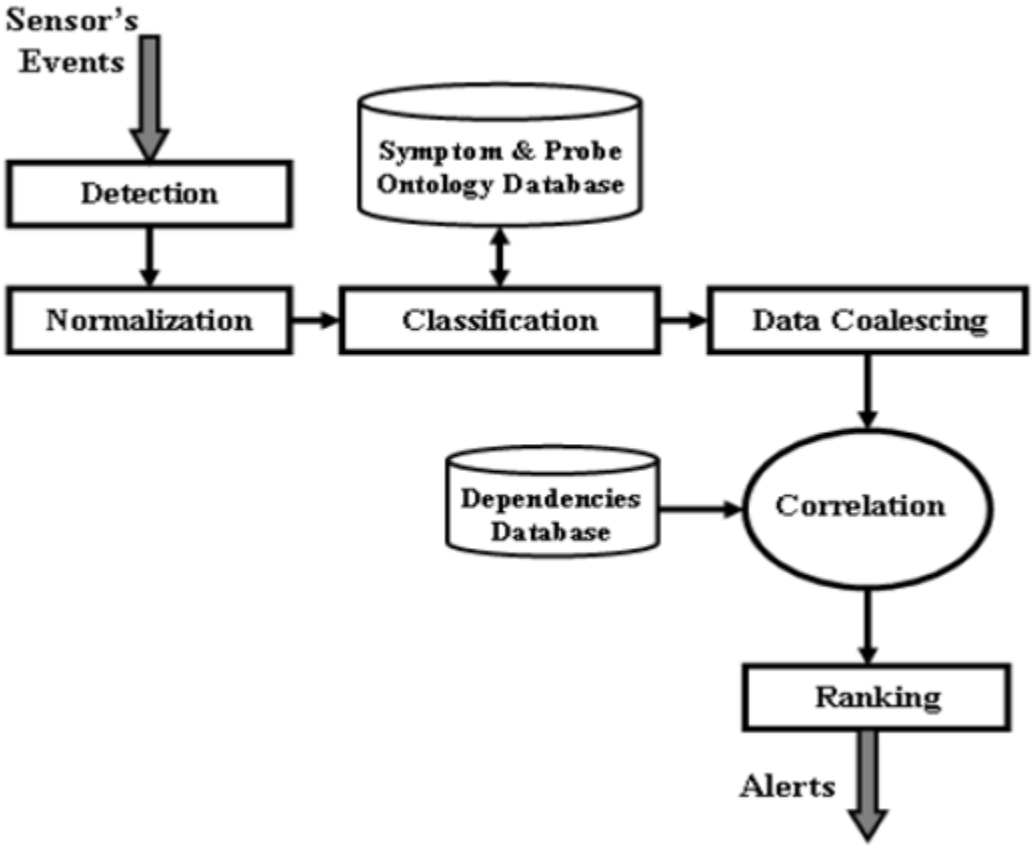


Figure 3: Correlation Process for weight-based approach

- **Detection** : Distributed probes and detection mechanisms are used to track different attack symptoms of an SQLIA.
- **Normalization** : Every detected symptom is recorded and normalized into a standardized format, and is also augmented with additional information such as time-stamps, source address of attacker, etc.
- **Classification** : Here, the symptoms are aggregated into categories depending on a number of different parameters.
- **Data coalescing** : Events generated by different probes detecting the same symptom are merged into a single event.
- **Correlation** : It receives the classified symptoms and correlates them by using various collaboration rules.
- **Ranking** : Once the correlation succeeds, a decision is taken whether the current observations correspond to malicious activities with respect to the system mission.

In general, the 'system mission' represents the major objectives of the detection process pursued by the security administrator. In this context in particular, the system mission for a SQLIA attack typically consists in avoiding unauthorized access to the backend database and any sensitive information.

Steps involved

1) **Detection** : The use of multiple heterogeneous and distributed probes potentially improves the detection performance through the generation of different perspectives of the same security incident. For example, the length of the query attributes, the error code generated by the database server, or the size of the pages returned by the web server could all be used as probes to detect symptoms of an SQLIA. Each probe uses a detection model that allows it to assign a probability value, called an 'anomaly score' (AS), which reflects the probability of the occurrence of the given anomaly with regards to an established profile in keeping with the system mission. Based on this value the evaluated feature is either classified as a potential attack's symptom or as normal.

2) **Normalization**: Since each probe can provide varied security information with differing representations or formats, a process of symptom normalization into a common format is imperative. On the basis of a specific mapping scheme, several attributes are associated with each event to aid the normalization process, such as the identifier of the probe, the symptom identifier, the source and the target of the attack, the start/end times of the symptom, and the anomaly score.

3) **Classification**: Classification aims to categorize symptoms. Categorization schemas must be defined to identify classes of symptoms in a prioritized, hierarchical manner with respect to the overall system mission. Symptoms may be categorized along several dimensions. For example, they could be divided into abuses, misuses, and suspicious

acts. *Abuses* represent actions which change the state of a system's asset, such as sensitive data or database schema, etc. These can further be divided into anomaly-based and knowledge-based abuses. The former represent anomalous behaviors (unusual application load, anomalous input requests); the latter are based on the recognition of signatures of previously known attacks (e:g:, brute force attacks). *Misuses* represent out-of-policy behaviors in which the state of the components are not affected (e:g:, authentication failed, failure queries). *Suspicious acts* are not policy violations of any kind but are merely events of interest to the probes (e:g:, commands which provide information about the state of the system).

4) **Data coalescing:** In order to avoid multiple messages referring to the same physical symptom from being generated, events that represent the independent detection (by different probes) of the same symptom occurrence are coalesced to a single event. When two symptoms are merged, the resulting event replaces the constituent events, and will be considered for matching with subsequent events. In particular, the resulting event presents an AS equal to the sum of the ASs of each of them.

5) **Correlation:** In this phase, the different symptom classes are correlated using one of a number of different correlation rules, and a meta-alert consisting of the correlated symptoms is generated. Researchers have proposed several alert correlation techniques and analysis processes. For example, a correlation rule that aggregates symptoms based not only on the impact they have on the system mission but also on their temporal proximity. Impact analysis requires a previous modelling of the relationships between symptoms and mission, which could either be determined through extensive experience and experimentation or through a heuristic-based technique.

6) **Ranking:** In order to reduce the effort required to analyze the volume of generated alerts, an approach based on the confidence of the meta-alerts is adopted. Assuming that S (k) = {fs1; s2; :::; sn} is the set of correlated symptoms during the time window k, the confidence is the probability that the meta-alert represents malicious actions with respect to the system mission.

4.2.1 Detection Approach

In order to detect SQLIA symptoms, anomaly detection models are adopted. They allow to assign a probability value (anomaly score) to the generated events, which reflects the probability of the occurrence of the given anomaly with regards to an established profile. The typical features used to detect symptoms are :

- **Character Distribution (CD**) : Typically, SQLIAs present a number of characters that are repeated many times and hence the character distribution can be highly anomalous. Therefore, an anomaly detection model is used to capture the

concept of 'normal' query attributes and flag any attempt at SQL injection based on the character distribution. During the training phase, for each HTTP GET and POST request, the query section is extracted, and the relative frequency of each character in the attributes is computed. Then the characteristics of normal character distribution are approximated by the average of all character distributions (the sum of the distributions is divided by the number of requests). The estimated frequencies are sorted and grouped, and any input query differing in its distribution of characters is marked with a corresponding Anomaly Score (AS).

- **Query Length (QL)** : The lengths of the inputs given in the different fields of a form that is part of a web request can be used to detect anomalous behaviors. Generally, the lengths of the query attributes do not vary much among requests associated with the same web application. However, this behavior may show considerable deviations during SQLIA's. For example, in UNION attacks, the attacker injects a statement of the form "UNION <injectedquery>", which changes the length of the query attribute quite significantly. A model is adopted which statistically estimates an approximation of the query's attribute length and detects suspicious inputs that significantly deviate from the observed normal behaviour.
- **Queries Failed (QF)** : SQLIAs that execute many queries on a particular database table could show up as an anomalous high rate of queries failed with respect to the normal behaviour. An operational model is adopted to estimate abnormal rate of queries failed with respect to the normal profile. This is considered over a fixed slicing time window.
- **Web Response (WR)** : The size of the page generated by the web server when under SQLIA can vary significantly from the size of the corresponding page during normal execution. For example, the web server could generate a web page which contains an error message, whose size is quite different compared to the normal response. In particular, it can be safely assumed that the size of the page generated against the same request does not vary by much and any such anomaly can be flagged as a symptom of an SQLIA. During the training phase it is necessary to estimate the mean and the variance of generated page for each web page directly reachable by the user.

4.2.2 Performance

Weight-based correlation approach for SQLIAs detection allows the system to assign a higher level of confidence to the alerts collected by multiple security probes, located at different architectural levels, so as to achieve a higher probability of spotting an intrusion. In comparison, the other methods are based on a single data source or on multiple data sources, but located at a single architectural level, and

are hence not as comprehensive. Weight-based approach is seen to give a very good performance in detecting a majority of both false positives and false negatives.

The injection attacks of the UNION type are very efficiently detected by the Query Length detection while the Tautology attacks are very well detected by the Character Distribution detection. Thus, assigning appropriate weights to these two detection probes could lead to the minimization of false positives. Also, a feedback learning technique could be used whereby the false positives once recorded can be avoided the next time by modifying the weights to generate better anomaly scores.

4.3 Preventing SQL Injection in Stored Procedures

4.3.1 Basics

This model has been proposed by Ke Wei, M. Muthuprasanna and Suraj Kothari of the Dept. of Electrical and Computer Engineering at Iowa State University in the research paper titled "*Preventing SQL Injection Attacks in Stored Procedures"*[2]. It essentially consists of two parts, a static analysis phase wherein an SQL graph is created to determine which inputs have to be checked for optimal execution time, and a run-time analysis wherein a FSA is used to determine whether a run-time query is clean from SQL injection attacks.

4.3.2 SQL Injection in stored procedures

Stored procedures, contrary to popular belief, are vulnerable to SQLIA's. Shown below is an example that illustrates how an attacker can exploit vulnerabilities in a stored procedure, to gain illegitimate access to the system as well as the network resources.

A stored procedure is an operation set that is precompiled and saved. Typically, stored procedures are written in SQL. Since stored procedures are stored on the server side, they are available to all the client instances. Once a stored procedure is modified, all clients automatically get the new version.

Shown below is a sample stored procedure that accepts 'Name' and 'Passwd' as user inputs in a variable length string format.

```
1. CREATE PROCEDURE [EMP].[RetrieveProfile] @Name varchar(50),
   @Passwd varchar(50)
2. WITH EXECUTE AS CALLER
3. AS
```

```
4. BEGIN
5. DECLARE @SQL varchar(200);
6. ...
7. SET @SQL='select PROFILE from EMPLOYEE where ';
8. ...
9. IF LEN(@Name) > 0 AND LEN(@Passwd) > 0
10. BEGIN
11. ...
12. SELECT @SQL=@SQL+'NAME="'+@Name+"' and ';
13. SELECT @SQL=@SQL+'PASSWD="'+@Passwd+"";
14. ...
15. END
16. ELSE
17. BEGIN
18. ...
19. SELECT @SQL=@SQL+'NAME="Guest"';
20. ...
21. END
22. ...
23. EXEC(@SQL)
24. ...
25. END
```

Figure 4: Stored Procedure vulnerable to SQL-Injection

An interesting observation to be made from the code sample shown above is that there is an EXEC system function which allows the user to dynamically build a SQL statement as a string and then execute it. This feature is supported in most business database products. Such dynamically constructed SQL statements provide great user flexibility (because database object names can be passed at runtime). However, they face a great threat from SQLIAs. The process of building an SQL statement could be used by the attacker to change the original intended semantics of the SQL statement.

If the stored procedure in Code 1 is called with no values for @Name and @Passwd variables, the following query would get executed: **select PROFILE from EMPLOYEE where NAME='Guest'**

When user inputs are provided for @Name and @Passwd, the following query would get executed: **select PROFILE from EMPLOYEE where NAME='name' and PASSWD='passwd'.**

In this scenario, suppose a user gives input for variable @Name as "' OR 1=1 --" (SLQ injection attempt) and any string, say "null", for the variable @Passwd the query would

take the form: **select PROFILE from EMPLOYEE where NAME=" or 1=1 –-' and PASS='null'.**

The characters "––" mark the beginning of a comment in SQL, and everything after that is ignored. The query as interpreted by the database is a tautology and hence will always be satisfied, and the database would return information about all users. Thus an attacker can bypass all the authentication modules in place and gain unrestricted access to critical data on the web server.

4.3.3 Proposed Solution

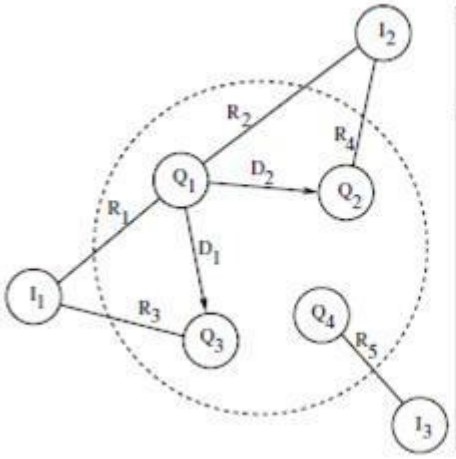
The proposed model is an SQL-Injection Attack prevention technique that addresses a majority of the different types of SQLIAs as discussed in previous sections. The model works by combining a static analysis along with runtime validation. "The basis of such a technique is that the control flow graph of the stored procedures can be represented as an SQL-graph which indicates what user inputs the dynamically built SQL statements depend on". By using an SQL-graph, the number of SQL queries that need to verified at runtime for SQLIA can be drastically reduced in most cases. This is done by selecting the smallest subset of queries that encompass all the string input parameters to the stored procedure. During runtime, a Finite State Automaton(FSA) is constructed from the EXEC(@SQL) procedure call and the is compared with the runtime query along with its associated inputs to determine if there is a change in the structure of the query, in which case it is flagged as an SQLIA.

Static Analysis

In the static analysis of a stored procedure, a stored procedure parser is used that retrieves the raw text of the procedure for analysis. Then, all the queries in the stored procedure that contain dynamic SQL constructs are identified and are saved as query nodes, with the inputs they depend on saved as specific input nodes. Once this is done, an SQL graph is constructed from the control flow of the stored procedure, which compares the input parameter dependencies of each query containing dynamic SQL, and finally selects the subset of all the queries which together contain as parameters the entire set of parameters used by the queries containing dynamic SQL statements. Only this subset of queries needs to be monitored at runtime, which avoids a lot of redundant computation that would otherwise have had to be done if this mechanism weren't used. This lowers the overall runtime execution time complexity and ensures better efficiency of the algorithm.

SQL-Graph Representation

It is possible for a stored procedure to have more than one EXEC(@SQL) statement. Not all the EXEC(@SQL) statements would depend on the user inputs. Only those which need the user inputs to complete the SQL statements are potentially vulnerable to SQLIA. However, the number of such statements in practical applications might yet be very large, and hence a mechanism is required to optimize the number of comparisons required to ascertain the legality and validity of an input parameter at runtime. An SQL graph is used in this model to achieve precisely that, an enhanced time and execution complexity and better overall algorithm efficiency. A sample SQL-Graph[2] is shown in the figure below.



Q_1, Q_2, Q_3, Q_4 I_1, I_2, I_3	SQL Queries User Inputs
R_1	$I_1 \in Q_1$
R_2	$I_2 \in Q_1$
R_3	$I_1 \in Q_3$
R_4	$I_2 \in Q_2$
R_5	$I_3 \in Q_4$
D_1	$Q_3 \subset Q_1$
D_2	$Q_2 \subset Q_1$

Figure 5: SQL Graph representation

The SQL-graph shown in the figure above represents 4 different SQL queries (EXEC statement hotspots) in the stored procedure as nodes and 3 different user inputs. If a particular user input (I) is used in a SQL query (Q), the relationship (R) between the two nodes is indicated by an undirected link between the 2 nodes. The SQL-graph also considers dependencies between queries. Dependencies (D) in the SQL-graph are shown as directed arrows that point from one SQL query to another SQL query such that the user inputs used by the former is a proper superset of the user inputs used by the latter. For SQL queries that use the same set of user inputs, one of them is arbitrarily chosen as a representative query and is made to point to the others.

The concept of directed dependencies given by an SQL-graph is used to reduce the runtime scanning overhead by restricting the number of queries that need to be scanned along any execution path that is taken in the stored procedure. SQL queries that do not use user inputs are not included in the SQL-graph. Only the SQL queries that are exposed to the user inputs in some form or the other (string manipulations included) are included in the SQL-graph representation. Once all the query nodes are added to the graph, the input dependencies are identified and corresponding query dependencies (based on input set closure) are determined. If a user input does not

cause any SQLIA in one query, it means that it conforms well to the SQL query semantics as defined by the SQL language. Then it is implicitly acknowledged that the same input in any other query would also not cause an SQLIA. If this knowledge is not exploited, a number of redundant verifications of the same user input in multiple SQL queries in the SQL graph would ensue. The directed dependency in the SQL-graph tells us which SQL queries are supersets of which other SQL queries in the SQL-graph. It would suffice to check only those SQL queries during runtime validation for SQLIA. The last step of the static analysis phase is the determination of this minimal set of queries that encompass all the inputs required, and storing that set of queries on the hard drive, to be fetched and analyses during runtime to scan for SQLIA's.

The SQL-graphs are constructed statically, and hence might need to be constantly updated to reflect any changes in the code made by the programmer at any point of time. To facilitate easy modification of the code by the developer in a transparent manner, there is an option for SQL Graph modification where the developers simply need to use the turn-off option to remove all the instrumentation and the corresponding SQL-graphs, alter the code as desired and then rebuild these elements again.

Runtime Analysis

During runtime, before the *EXEC()* function in the stored procedure is called, the *SQLIACHECKER*() function will identify the user input by the current session ID and build a finite state automaton. Then, the SQL statement with user inputs filled in is compared against the corresponding FSA to check for validity. If the user inputs result in the dynamically generated SQL queries not conforming to the semantics of the intended SQL queries as in the FSA, then they are flagged as SQLIAs, else they are passed through. The figure shown[2] below illustrates two cases; one where an SQLIA is not caused and the query is passed through i.e. it matches the automaton construct, and another where an SQLIA has been caused and hence gets rejected as a potentially malicious query.

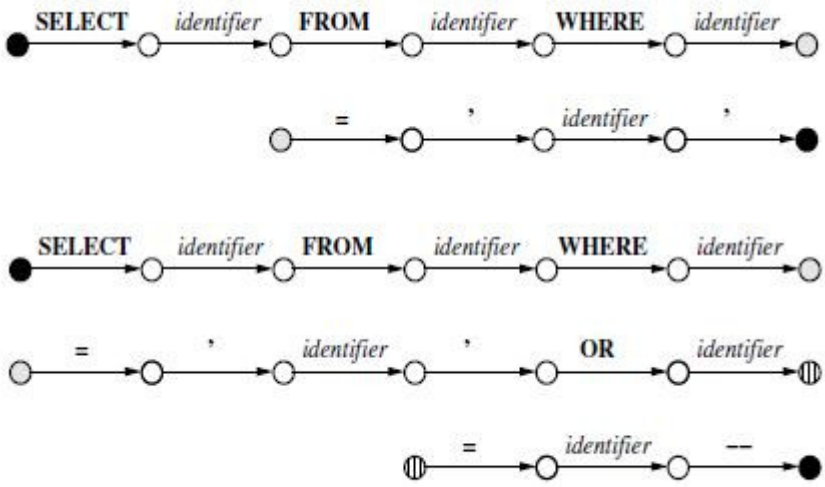


Figure 6: SQLIA Detection: SQL-FSM Violation

The literals (token encountered in the FSA) along both the original structure and the user inputs included structure, as one traverses from the *Start* node to the *End* node, should be exactly identical. The other check that can be enforced is that the length of the finite state automaton chain for a particular instance is exactly the same for the original one and the one with the inputs added. Thus SQLIAs employing tautology attacks and those injecting additional statements can be effectively captured by this technique. The case where alternate encoding like *URL Encoding*, *UTF-8* etc. are used by the attackers can also be addressed using this technique by enforcing that the runtime validation occur only after all the user inputs have been converted to a single encoding format as interpreted by the SQL Engine in the database server.

For each of the SQL queries listed in the CHECKLIST(), an FSA comparison is made and if all of these are found to conform to the statically constructed FSA's, then we can ascertain the fact that all the input parameters required which might have been vulnerable to SQLIA's are indeed legal and the stored procedure is now safe to be forwarded to the database for execution.

4.4 MUSIC: Mutation Based SQL Injection Checking

This model has been proposed by Hossain Shahriar and Mohammad Zulkernine of the School of Computing at Queen's University, Ontario, Canada in the research paper titled

"*MUSIC:Mutation-based SQL Injection Vulnerability checking"*[1]. In this work, they have applied the concepts of mutation based testing to SQLIA verification. Mutation is basically a fault-based testing technique, where a program is injected with faults according to rules known as mutation operators. The resultant state is called a mutant, and is either killed by the test cases, or remains alive. Depending on this state of the mutant, it can be determined whether the test is successful or not, which is dictated by the overall system objective.

4.4.1 Mutation operators

This model proposes nine mutation operators divided into two categories. The first category consists of four operators that inject faults into the WHERE conditions (WC) of the SQL queries. The second category consists of five operators that inject faults into the database API method calls (AMC). A summary of the proposed mutation operators is provided in the table shown below. The mutant killing criteria shown in the table[1] are described in the following section.

Cat.	Operators	Description	Killing criteria
WC	RMWH	Remove WHERE keywords and conditions.	C_0
	NEGC	Negate each of the unit expression inside where conditions.	C_1
	FADP	Add parentheses in where conditions and prepend "*FALSE AND*" after the WHERE keyword.	C_2
	UNPR	Unbalance parentheses of where condition expressions.	C_3
AMC	MQFT	Set multiple query execution flags to true.	$C_4 \| C_5 \| C_6$
	OVCR	Override commit and rollback options.	C_4
	SMRZ	Set the maximum number of record returned by a result set to infinite.	C_6
	SQDZ	Set query execution delay to infinite.	C_7
	OVEP	Override the escape character processing flags.	C_5

Figure 7: The Proposed Operators in MUSIC

4.4.2 Mutant Killing Criteria

The eight distinguishing or killing criteria proposed by the model are as shown in the following table. A test case that satisfies any of the eight criteria will be able to kill a generated mutant. Let *I* and *M* be the intended and its corresponding mutated query, respectively. Let us assume that the two queries use tables having *N* number of records in total ($N > 1$) and let *n1* and *n2* be the set of records selected on execution of *I* and *M*, respectively. The criterion *C0* given in the table distinguishes *I* and *M*, if either (i) the cardinality of the intersection between *n1* and *n2* is zero or *N*; or (ii) the cardinality of the union of *n1* and *n2* is greater than *N*. Similarly, criterion *C1* distinguishes between *I* and *M*, if the cardinality of the intersection between *n1* and *n2* is not zero. *C2* differentiates between *I* and *M*, if the cardinality of the intersection of *n1* and *n2* is greater than zero. For understanding the working of criterion *C3*, let us assume *s1* represents an application state, in which query *I* runs successfully and query *M* results in a syntax error and *s2* represents the opposite state of *s1* (*i.e.*, *M* runs successfully and *I* generates syntax error message). The criterion *C3* is used to distinguish between *I* and *M* based on the observation that $s1 \neq s2$. The remaining criteria for killing mutants may be understood similarly using the table[1] shown below.

Name	Distinguishing criteria
C_0	$(\|n_1 \cap n_2\| = 0) \,\|\|\, (\|n_1 \cap n_2\| = N) \,\|\|\, (\|n_1 \cup n_2\| > N)$
C_1	$\|n_1 \cap n_2\| \neq 0$
C_2	$\|n_1 \cap n_2\| > 0$
C_3	$s_1 \neq s_2$
C_4	$(i_1 \neq i_2) \,\|\|\, (d_1 \neq d_2) \,\|\|\, (u_1 \neq u_2) \,\|\|\, (o_1 \neq o_2)$
C_5	$p_1 \neq p_2$
C_6	$\|n_2\| > \|n_1\|$
C_7	$(t_1 > T \;\&\&\; t_2 < T) \,\|\|\, (t_1 < T \;\&\&\; t_2 > T)$
I: Intended query; *M:* Mutated query. n_1: The record set selected by *I*. *n2:* The record set selected by M. s_1: State where *I* runs successfully and *M* generates error message. s_2: State where *M* runs successfully and *I* generates error message. i_1: # of records inserted by *I*. i_2: # of records inserted by *M*. d_1: # of records deleted by *I*. d_2: # of records deleted by *M*. u_1: # of records updated by *I*. u_2: # of records updated by *M*. o_1: # of database objects created by *I*. o_2: # of database objects created by *M*. p_1: # of external objects created by database by *I*. p_2: # of external objects created by database by *M*. t_1: Time elapsed to execute *I*. t_2: Time elapsed to execute *M*. *T:* Default query execution timeout of application.	

Figure 8: Mutant Killing Criteria

4.4.3 Details of Mutation Operator

A sample database table named '*tlogin'* is shown below. The table has three columns named '*id'*, '*uid'*, and '*pwd'*, which represent the unique ID number of a user, his login ID, and his login password, respectively. Let us assume that an intended query written by a programmer is *"select id from tlogin where uid='" + userid +"'".* Here, *userid* is a string variable that receives the user supplied '*userid'* and then becomes part of the query generation process without any filtering operation. The *'tlogin'* table[1] is as shown:

id (numeric)	uid (varchar)	pwd (varchar)
1	aaa	aaa
2	bbb	bbb
3	ccc	ccc

Figure 9: Table tlogin

Remove SQL where conditions (RMWH). The RMWH operator removes the WHERE conditions of SQL queries, which results in the selection of all the rows from the particular table. The generated mutants are killed by test cases that satisfy the criterion *C0*. SQLIA's of tautology or UNION type will be able to kill the generated mutants. The operator is applicable for SELECT, UPDATE, and DELETE type queries.

T	Intended query (*I*)	Mutated query (*M*)	n_1	n_2	St.
aaa	select id from tlogin where uid='aaa'	select id from tlogin //ΔRMWH	{1}	{1, 2, 3}	L
' or 1=1 --	select id from tlogin where uid='' or 1=1 --	select id from tlogin //ΔRMWH	{1, 2, 3}	{1, 2, 3}	K
' union select 20 --	select id from tlogin where uid='' union select 20 --	select id from tlogin //ΔRMWH	{20}	{1, 2, 3}	K
T: Test case. St.: Status of mutant. L: Live. K: Killed.					

Figure 10: Example Applications of RMWH

The table[1] above shows four example applications of the RMWH operator. The first row shows the intended query (*I*) and the mutated query (*M*) with the test cases *'aaa'*. Since *'aaa'* is a valid *uid* for the table shown above, it can be validated as not containing any SQLIA. Execution of *I* and *M* give the result sets *n1* and *n2*, which are row numbers {1} and {1, 2, 3}, respectively. The intersection and union of the two sets is {1} and {1, 2, 3}, respectively. The cardinalities of intersection and union are 1 and 3, respectively. Therefore, the test case does not satisfy the criterion *C0*, and the mutant remains *live*. However, the second and third test cases, which contain tautology type *("' or 1=1 –")*

and union type ("*' union select 20 --*") attacks, respectively, kill the corresponding mutants.

Negation of expression in WHERE conditions (NEGC). The NEGC operator negates the unit expressions (*e.g.*, *uid='aaa'* to *uid!='aaa'*) present in the WHERE condition of an SQL queries. The operator is applicable for SELECT, UPDATE, and DELETE type queries. The intersection of the two record sets (one selected by an arbitrary condition and the other selected by its negation) should be null, provided the semantic of the query does not change. This fact is taken advantage of to force the generation of attack test cases that will violate the observation (*i.e.*, satisfy the *C1* criterion). The table[1] shown below shows three examples of the NEGC operator, where the equal (=) operator in *I* is mutated to not equal (!=) in *M*. The first row of the table shows that *'aaa'* (non SQLIA test case) cannot kill the mutant. The mutant is, however, killed by the tautology (second row) and union attack test cases (third row), where the criterion *C1* is satisfied.

T	Intended query (*I*)	Mutated query (*M*)	n_1	n_2	St.
aaa	select id from tlogin where uid='aaa'	select id from tlogin where not uid!='aaa' //ΔNEGC	{1}	{2, 3}	L
' or 1=1 --	select id from tlogin where uid='' or 1=1 --	select id from tlogin where not uid!='' or 1=1 -- //ΔNEGC	{1, 2, 3}	{1, 2, 3}	K
' union select 20 --	select id from tlogin where uid='' union select 20 --	select id from tlogin where not uid!='' union select 20 -- //ΔNEGC	{20}	{1, 2, 3, 20}	K
T: Test case. St.: Status of mutant. L: Live. K: Killed.					

Figure 11: Example Application of NEGC

The remaining operators can be applied in a manner similar to the ones discussed above. The main objective of MUSIC is to detect SQL injection vulnerabilities rather than prevent them, the main concept being that if the mutant is alive then the code is safe, else if the mutant is killed then it indicates that there is an SQL vulnerability.

5.0 Key Observations

1. Stored procedures with dynamic SQL are prone to SQL injection attacks.
2. Dynamic SQL is inevitable when the system objects aren't know statically, for e.g. table names, column names, sort order.
3. SQL injection in dynamic SQL can be mitigated by using parameterized queries or prepared statements.
4. However there are situations when the concatenation of user inputs to generate the query is unavoidable. These situations are as listed below:-

   Large Number of insertion/deletion operations

   If there are large number of insertion and deletion operations between two executions of the same query, the query plan is bound to change for the two cases due to the large difference in the number of records. If a precompiled query or prepared statement is used then the query plan once made is cached and even if the number of records vary then the same old plan is used thereby affecting the performance of the system.

   Change in the structure of the table

   If there is a change in the structure of the table between two executions of the query, then the cached query plan will give sub-optimal solution. Hence compilation of the query between executions is necessary and hence concatenation needs to be used.

   Change in the type of index

   If the type of index changes between the executions of the query, it affects the cost of the query plan and it is necessary to compile the query between executions to obtain better performance.

   Number of parameters supported

   There exists a limit on the number of parameters that a parameterized query or a prepared statement can take. In practical situations and while using data mining or data warehousing there arise cases when the number of parameters exceeds that supported. Hence concatenation again is inevitable.

Having made the above mentioned observations, and after a careful analysis of the different models described, we noted that since both the weight-based symptom

correlation approach and the mutation based SQLI checking approach were focused on standalone query processing, and since a lot of research work has already been done in the area and lots of model for preventing SQLIA's proposed, it wouldn't be particularly rewarding for us to continue our research in that domain. However, we were very intrigued by the paper describing the model to prevent SQLIA's in stored procedures, and seeing how widely stored procedures are being used in commercial applications today both for performance and for security reasons, we found it prudent to delve into the realm of SQLIA prevention in stored procedures. Moreover, we found the misconception that merely using stored procedures in an application makes it safe from SQLIA's to be a very popular one, and this was something we sought to address through this project work.

Hence, we chose the injection prevention model proposed by Wei, Muthuprasanna et al. as the pertinent one considering the above mentioned factors, choosing it over the model proposed in the paper 'Secure Query Processing by Blocking SQL Injection' by Dibyendu Aich because of the relative time and space complexity issues, operating efficiency and model elegance of the former. Moreover, after seeing that there haven't been any published papers about the implementation of the model proposed, we decided that it would be a great topic to pursue our project work in. The rest of this thesis work contains the details of our implementation of the model presented in the paper 'Preventing SQL Injection Attacks in Stored Procedures' by Ke Wei, M. Muthuprasanna and Suraj Kothari, with a few divergences from the model during implementation. The programming specifics and the structures used therein were entirely designed by us.

6.0 Implementation Details

6.1 DFD's for Static Analyzer

6.1.1 Level 0

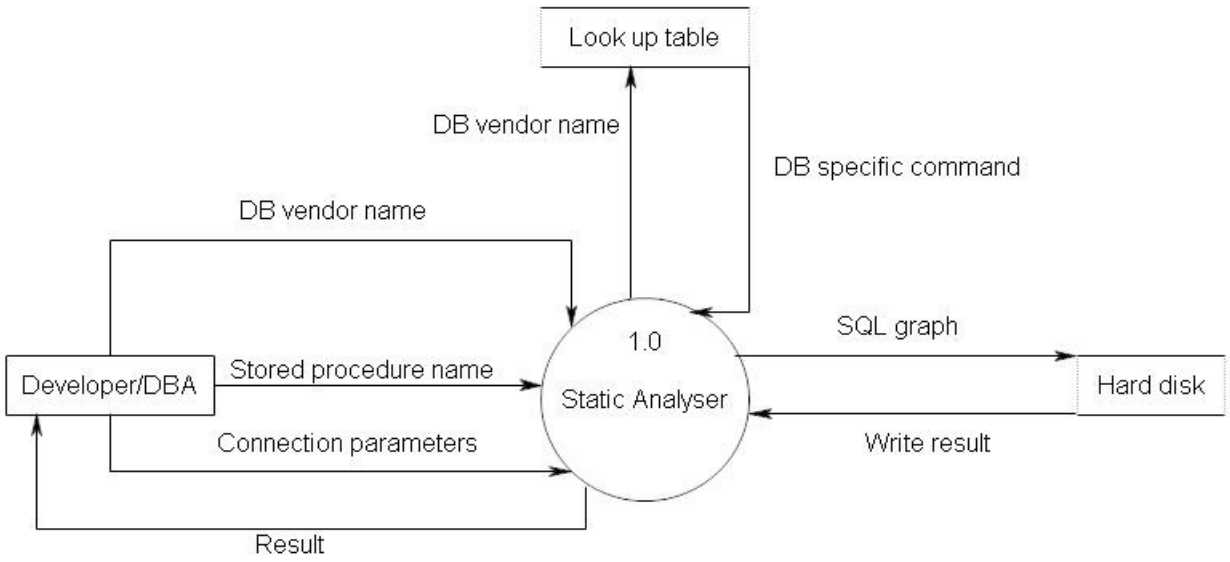


Figure 12: Level 0 DFD for Static Analyzer

6.1.2 Level 1

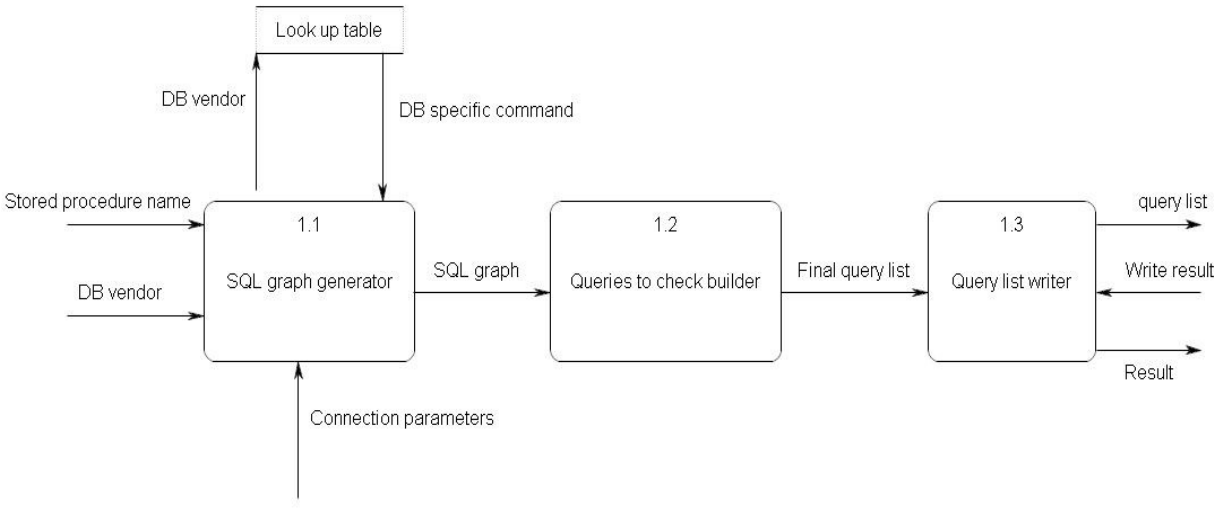


Figure 13: Level 1 DFD for Static Analyzer

6.1.3 Level 2

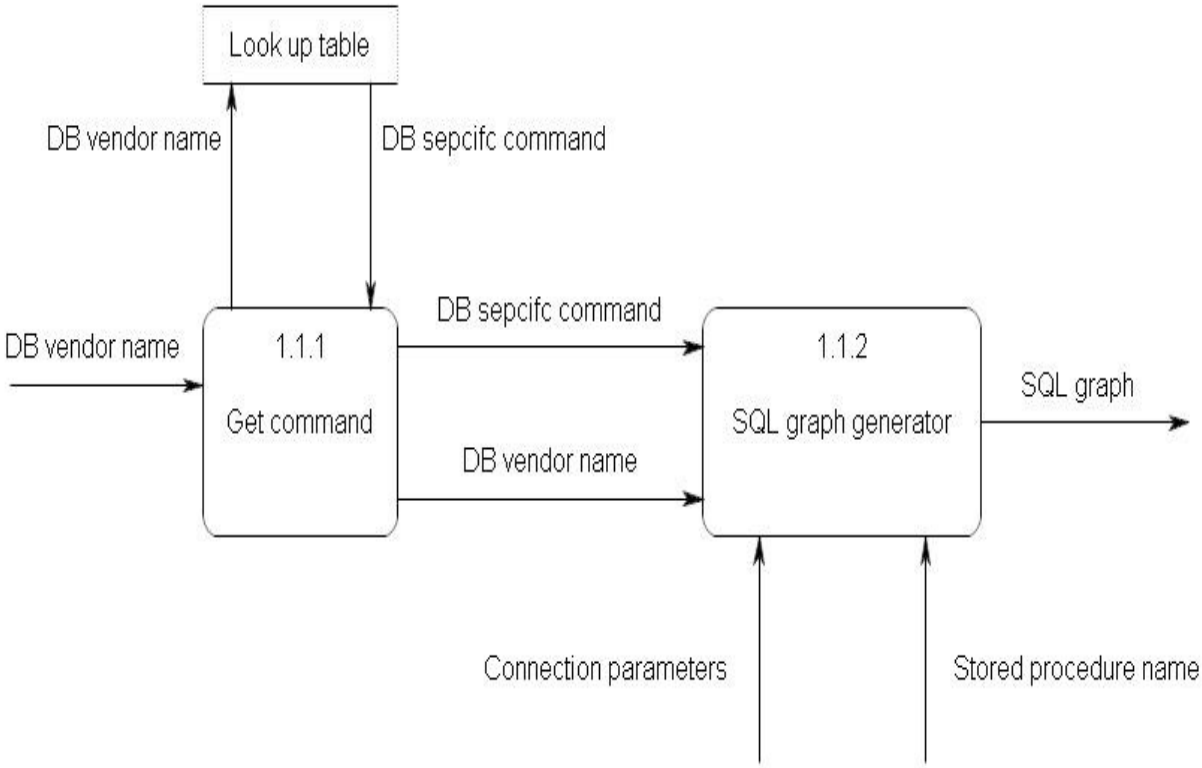


Figure 14: Level 2 DFD for Static Analyzer

6.2 DFD's for Runtime Analyzer

6.2.1 Level 0

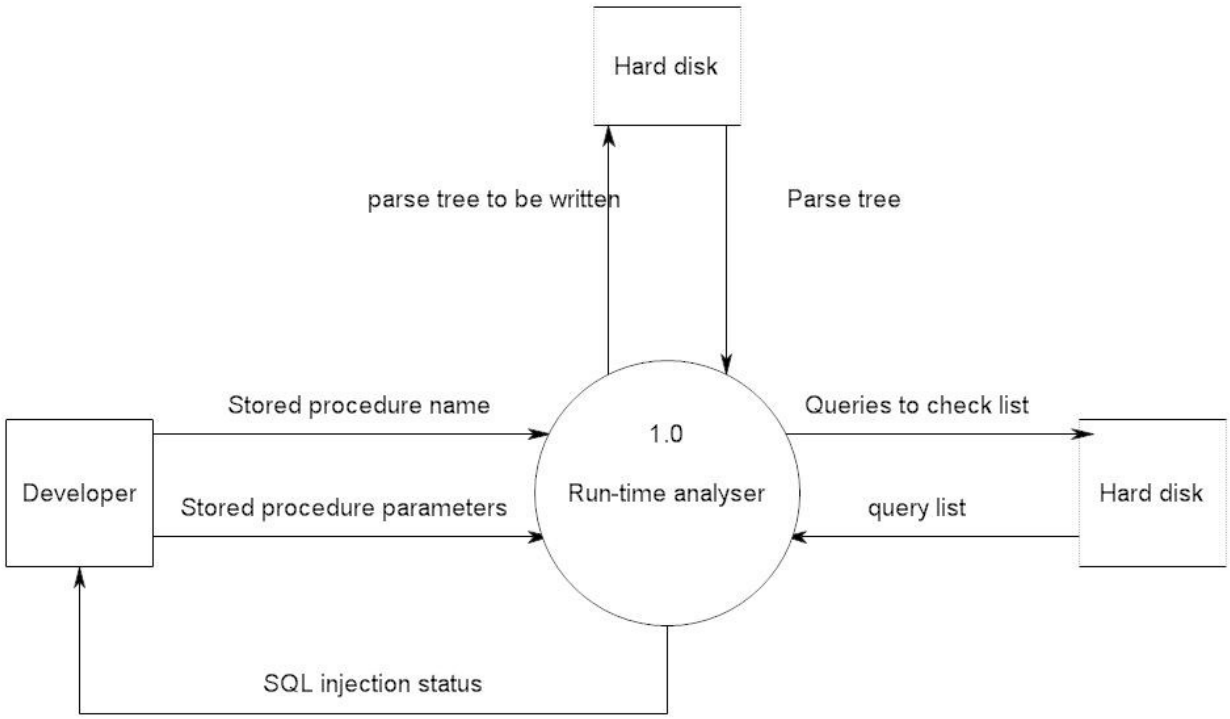


Figure 15: Level 0 DFD for Runtime Analyzer

6.2.2 Level 1

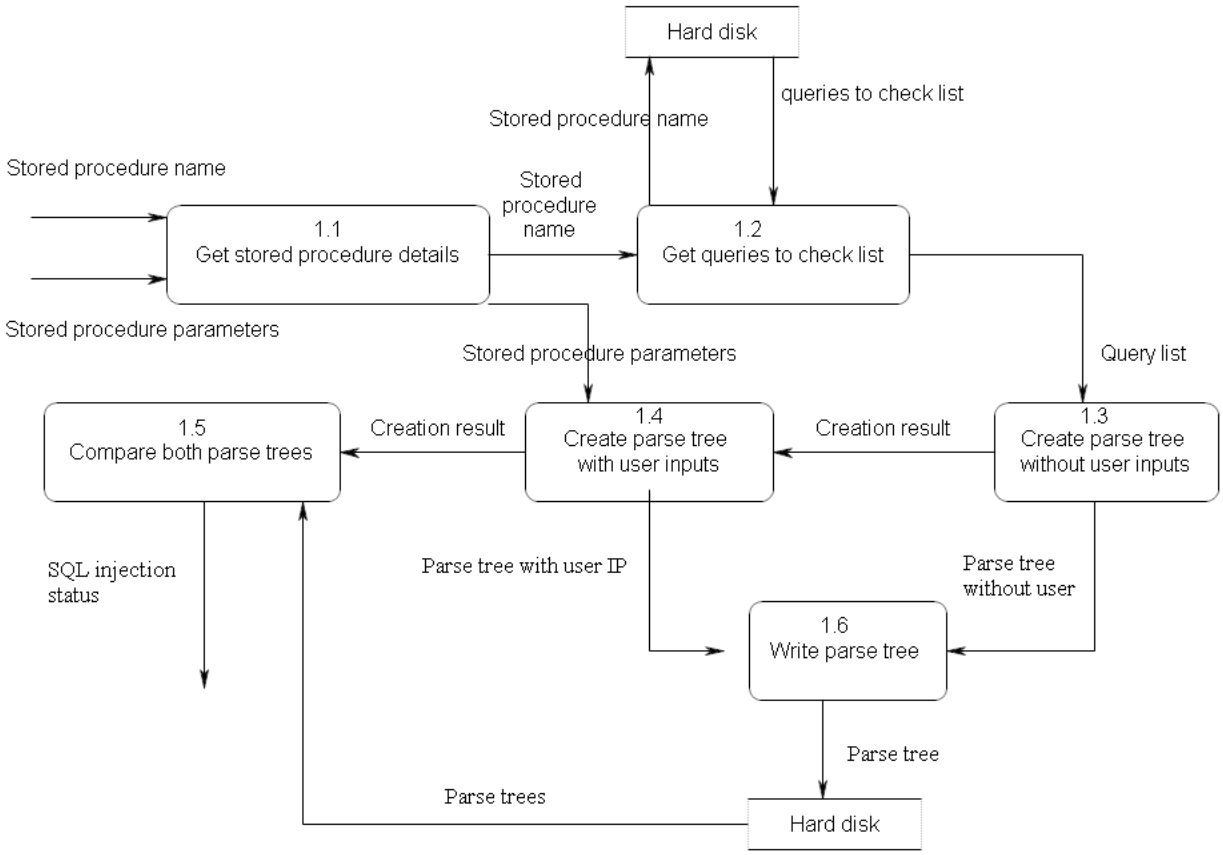


Figure 16: Level 1 DFD for Runtime Analyzer

6.3 Graphical user interface

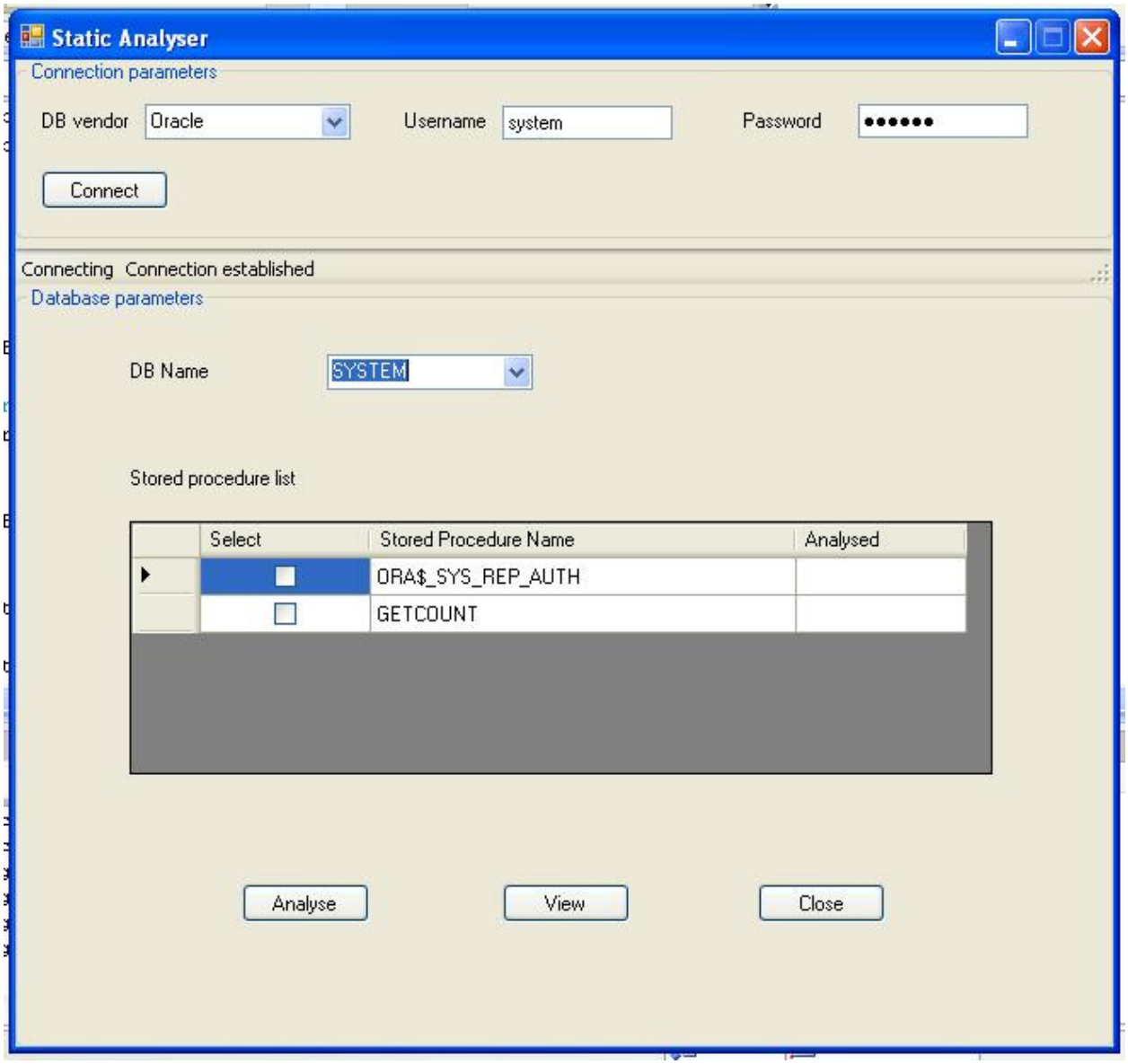


Figure 17: Main working window

Figure 18: View stored procedure text window

6.4 Structure Description and Technical Specifications

There are three basic data structures that are used in our implementation, namely:

1. Query Node:
   - Query string – To store the query string.
   - Input list – List of inputs on which the query depends.

2. Input Node:
   - Input variable name.
   - Query list – List of all queries that depend on this input
   - Position in parameter list – Position in the stored procedure parameter list

3. Graph Node
   - Query node list – List of all query nodes in the graph
   - Input node list – List of all input nodes in the graph
   - Query list – List of all queries that need to be checked for SQLIA

The software has been developed on the Microsoft .NET platform, using the C# language. During the static analysis, the list of queries to be checked is written onto the hard disk and is later fetched by the runtime analyzer to create parse trees and validate the queries. The parse trees are created using the SQL parser developed by Serge Gorbenko who is a senior software developer from Ukraine. This SQL parser is available at codeproject.com. At the preliminary stage, we have included support for the Oracle database. However, the implementation has been made generic enough to easily extend support to any database in the future.

6.5 Pseudo code

6.5.1 Static Analyzer

```
If string.contains(EXEC)
{
        Search for EXEC
        while(char!=" ")
        {
                check char after EXEC
                if("(")
                {
                        Push  "("  into stack
                        Make query node
                        Till stack is empty
                        {
                                If nextchar() = "+"
                                {
                                        Create input node
                                        While nextchar()! = "+"
                                        {
                                                Input.setname = nextchar()
                                                Query.text = nextchar()
                                        }
                                        Set input.value()
                                        Query.inputlist = inputnode
                                        Input.querylist = querynode
                                }
                                Else if nextchar() = "("
                                {
                                        Push into stack
                                        Query.text = nextchar()
                                }
                                Else if nextchar() = ")"
                                {
                                        Pop from stack
                                        Query.text = nextchar()
                                }
                                Else
                                        Query.text = nextchar()
                        }
                }
                Else
                {
                        Create new query node
```

```
                    While nextchar = " " or ";"
                    Temp=nextchar();
                    Find last definition of temp by reverse string matching
                    Extract the query text as in the previous case
                }
            }
        }
```

6.5.2 Runtime Analyzer

1. set storedProcedureName, storedProcedureParameters
2. set queriesToCheck:= graphNode.GetQueriesToCheck();
3. Foreach queryNode in queriesToCheck
    a. Set string q1=query without user inputs
    b. Set string q2=query with user inputs
    c. Set XMLTextReader rdr1=sqlParser.parse(q1)
    d. Set XMLTextReader rdr2=sqlParser.parse(q2)
    e. If(rdr1.LocalName==rdr2.LocalName)
        i. If(!(rdr1.LocalName=="") && !(rdr2.LocalName==""))
            1. If(rdr1.LocalName=="Tag")
                a. Set attributeName="Type"
            2. Else if(rdr1.LocalName=="Text")
                a. Set attributeName="Value"
            3. If(rdr1.AttributeCount!=0 && rdr2.AttributeCount!=0)
                a. If(!(rdr1.GetAttributes(attributeName)==rdr2.GetAttributes(attr ibuteName))
                    i. Flag as SQL injection
                    ii. Break;
    f. Else
        i. Flag as SQL injection
        ii. Break;
4. If(SQLInjection flag is set)
    a. Raise exception or return true
5. Else
    a. Return false

7.0 Results and Performance

7.1 Results

The static analyzer and the runtime analyzer were tested against various stored procedures. The stored procedures were built to work on the standard databases of the Oracle database. The standard database was the EMP database, with the tables 'emp','dept' and 'customer' used for testing the software. A sample stored procedures used for testing the software is as shown below. This procedure contains 12 query statements with dynamic SQL constructs, and depends on 5 user input parameters.

```
create or replace procedure CustomerTableTwelve(custName in varchar,custAdd in varchar,custCity in varchar,custState in varchar,custPhone in varchar) is

query varchar2(200);
begin
query:='select city from customer where name='||custName||'';
execute immediate query;
query:='select city from customer where name='||custName||' and address='||custAdd||'';
execute immediate query;
query:='select name from customer where city='||custCity||'';
execute immediate query;
query:='select creditlimit from customer where name='||custName||' and city='||custCity||'';
execute immediate query;
query:='select address from customer where name='||custName||' and phone='||custPhone||'';
execute immediate query;
query:='select name from customer where state='||custState||'';
execute immediate query;
query:='select custid from customer where state='||custState||' and city='||custCity||'';
execute immediate query;
query:='select repid from customer where phone='||custPhone||'';
execute immediate query;
query:='select name from customer where city='||custCity||' and state='||custState||'';
execute immediate query;
query:='select comments from customer where name='||custName||'';
execute immediate query;
query:='select custid,name from customer where address='||custAdd||'';
```

```
execute immediate query;
query:='select area from customer where name='||custName||'';
execute immediate query;
end CustomerTableTwelve;
/
```

The following table gives the summary of the testing results.

Type of injection	Successfully blocked	Number of false positives
Tautology	Yes	Nil
Union	Yes	Nil
Piggy backed	Yes	Nil
Logically incorrect query	Yes	Nil
End of line comment	Yes	Nil

Table 1: Summary of testing results

The runtime analyzer uses a **parse tree validation** to validate the structure of the queries. Since this method is sure to detect any changes in the structure of the queries, any type of SQL injection which changes the query structure can be detected using the implementation given in this thesis work.

7.2 Performance analysis

7.2.1 Static Analysis

Number of queries versus execution time

The performance evaluation for this case was done by varying the number of queries in the stored procedure while keeping the user input dependencies of each query constant at one dependency each. The execution times for the static analyzer were measured by varying the number of queries for 50 iterations and the average times were calculated. The graph shown below gives the relation between the number of queries in the stored procedure versus the average time the static analyzer takes to analyze the stored procedure. The graph is seen to show an increasing trend, as predicted, where higher the number of queries, higher is the execution time. However there is minimal overhead of using the static analyzer as can be seen from the average times in the table below. The average execution time for analysis of a stored procedure having 12 queries is just 3.7ms, which is a very reasonable compromise in most cases considering the enhanced level of security it provides.

Sr No	Number of queries	Average time in ms
1	1	1.61
2	3	2.5
3	6	2.97
4	9	3.2
5	12	3.7

Table 2: Execution time for different no. of queries for SA

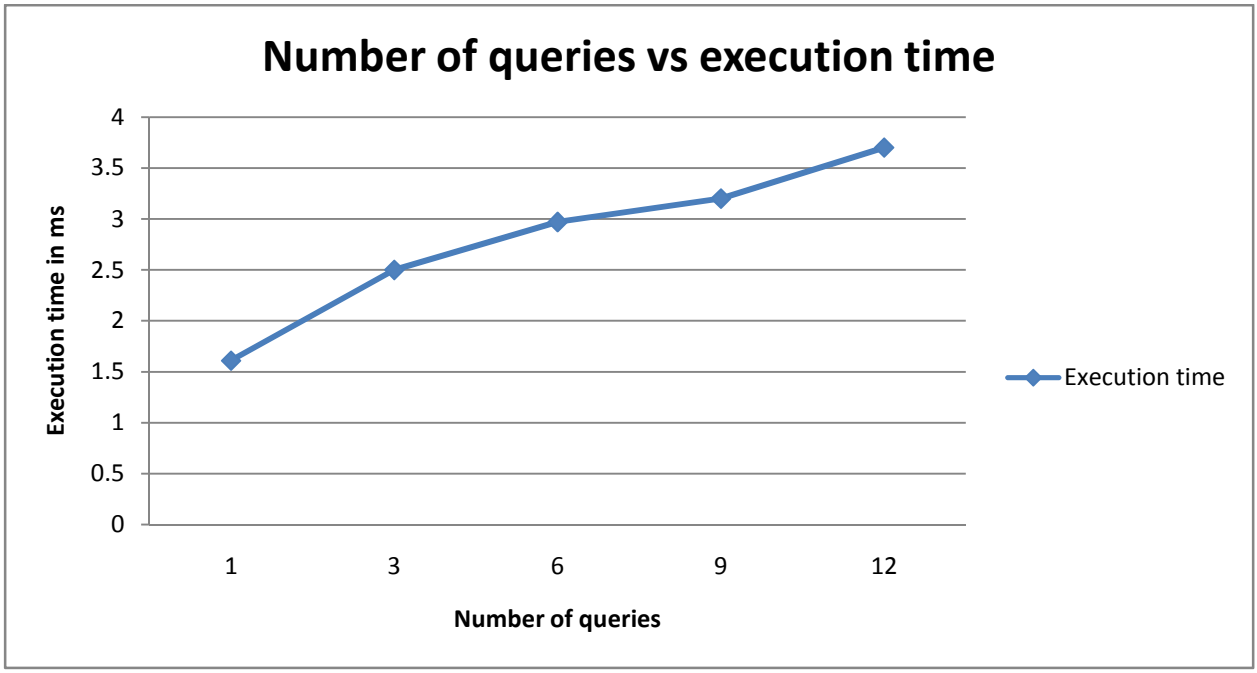


Figure 19: Graph: Number of queries vs execution time for SA

Number of dependencies versus execution time

The performance evaluation for this case was done by fixing the number of queries in the stored procedure to one but varying the number of user input dependencies in the query. The execution times for the static analyzer were measured by varying the number of queries for 50 iterations and the average times were calculated. The graph below shows the relation between the number of user input dependencies and the execution time of the static analyzer for the given stored procedure. As predicted, this graph also shows an increasing trend; higher the number of dependencies, higher is the execution time of the analyzer. The overhead isn't very significant, as can be seen from the table below. The execution time is 2.86 ms for a query with five user input dependencies.

Sr No	Number of dependencies	Average time in ms
1	1	1.71
2	2	1.93
3	3	2.2
4	4	2.42
5	5	2.86

Table 3: Execution time for different no. of dependencies for SA

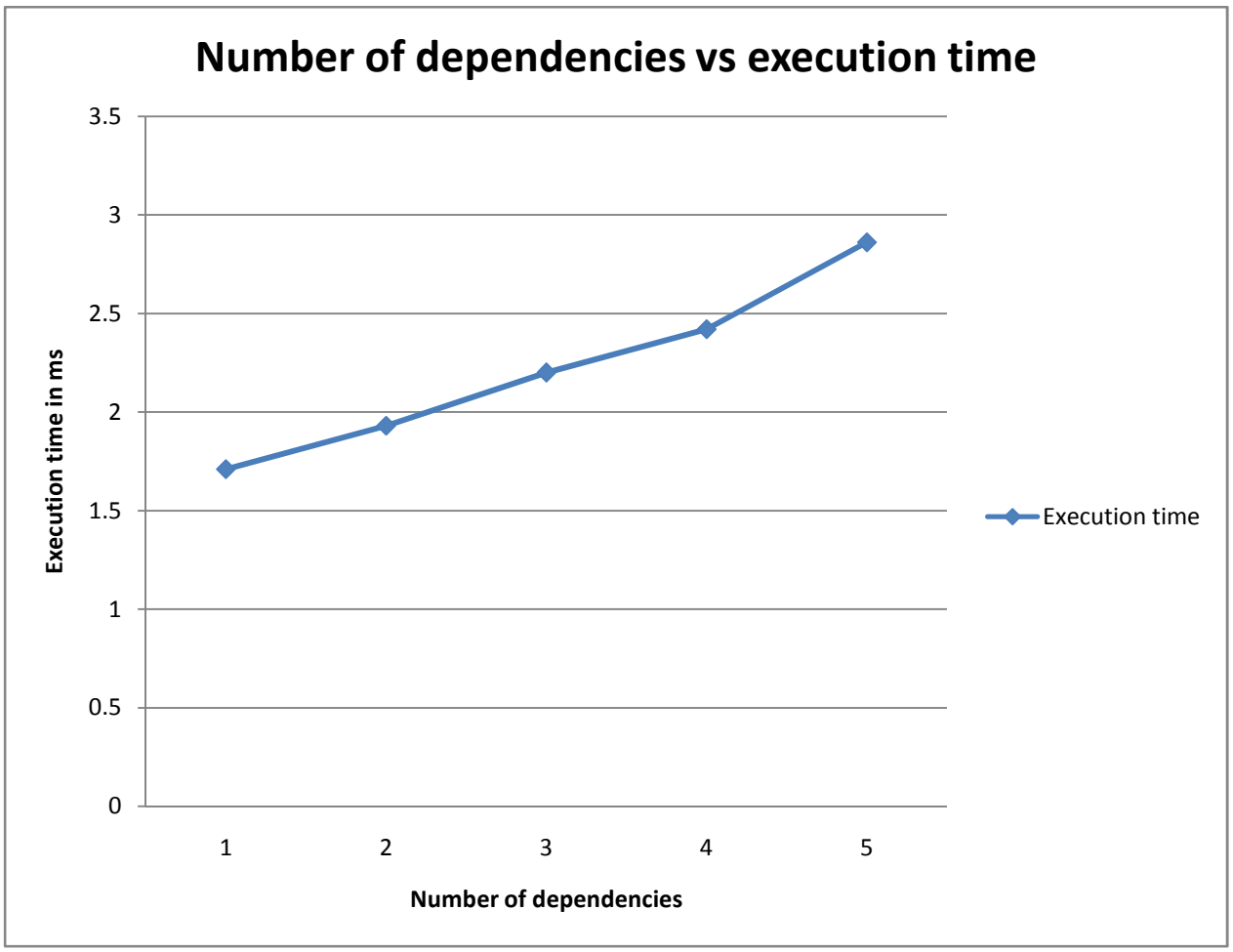


Figure 20: Graph: Number of dependencies vs execution time

7.2.2 Runtime Analysis

The performance of the runtime analyzer is an important consideration in this thesis, as the runtime analyzer incurs the overhead of checking the user inputs and the stored procedures for possibility of any SQL injection. Any application that uses this implementation would have some amount of overhead due to the runtime analysis. However the overhead is found to be quite reasonable within the scope of most application.

Number of dependencies versus execution time

The performance evaluation for this case was done by fixing the number of queries in the stored procedure to one and varying the number of user input dependencies in the query. The execution times for the runtime analyzer were measured by varying the number of queries for 50 iterations and the average times were calculated. Typical user inputs were given to the run time analyzer. The graph below shows the relation between the number of user input dependencies and the execution time of the runtime analyzer for the stored procedure. The nature of the graph is increasing i.e. higher the number of dependencies, higher is the execution time of the analyzer. The overhead is fairly reasonable, as can be seen from the table below. The execution time is 27 ms for a query with five dependencies.

Sr No	Number of dependencies	Average time in ms
1	1	14
2	2	15
3	3	20
4	4	24
5	5	27

Table 4: Execution time for different no. of dependencies for RA

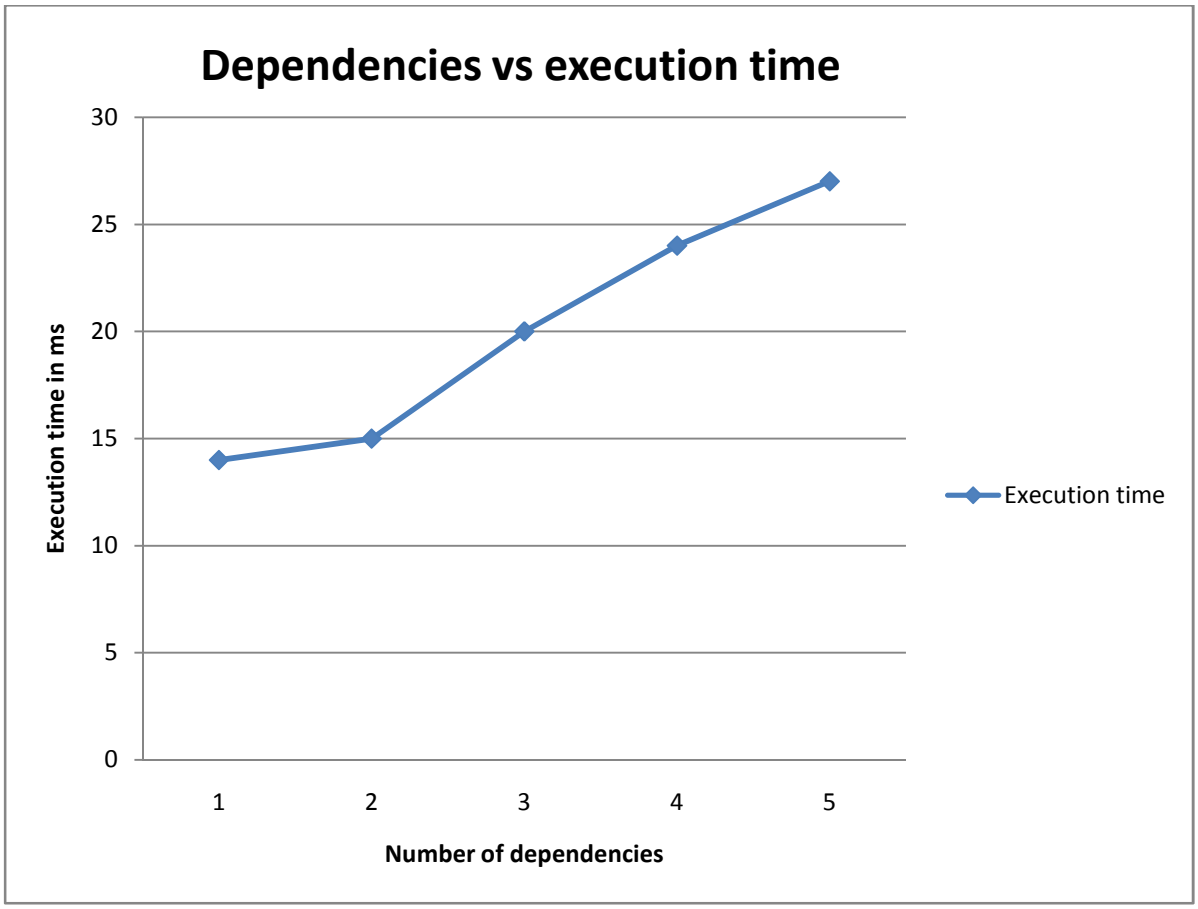


Figure 21: Graph: Number of dependencies vs execution time for RA

Number of queries versus execution time

The performance evaluation for this case was done by varying both the number of queries in the stored procedures as well as the number of dependencies of each query. As such this evaluation was divided into three categories namely queries with one dependency each, queries with two dependencies each and queries with three dependencies. In each category the number of queries in the stored procedures was varied and the execution time of the runtime analyzer was measured for 50 iterations and the average time was calculated. The graph in each category is increasing in nature; higher the number of queries higher the execution time.

Queries with one dependency each

Sr No	Number of queries	Average time in ms
1	1	12
2	2	24
3	3	32
4	4	50

Table 5: Execution time for queries with one dependency for RA

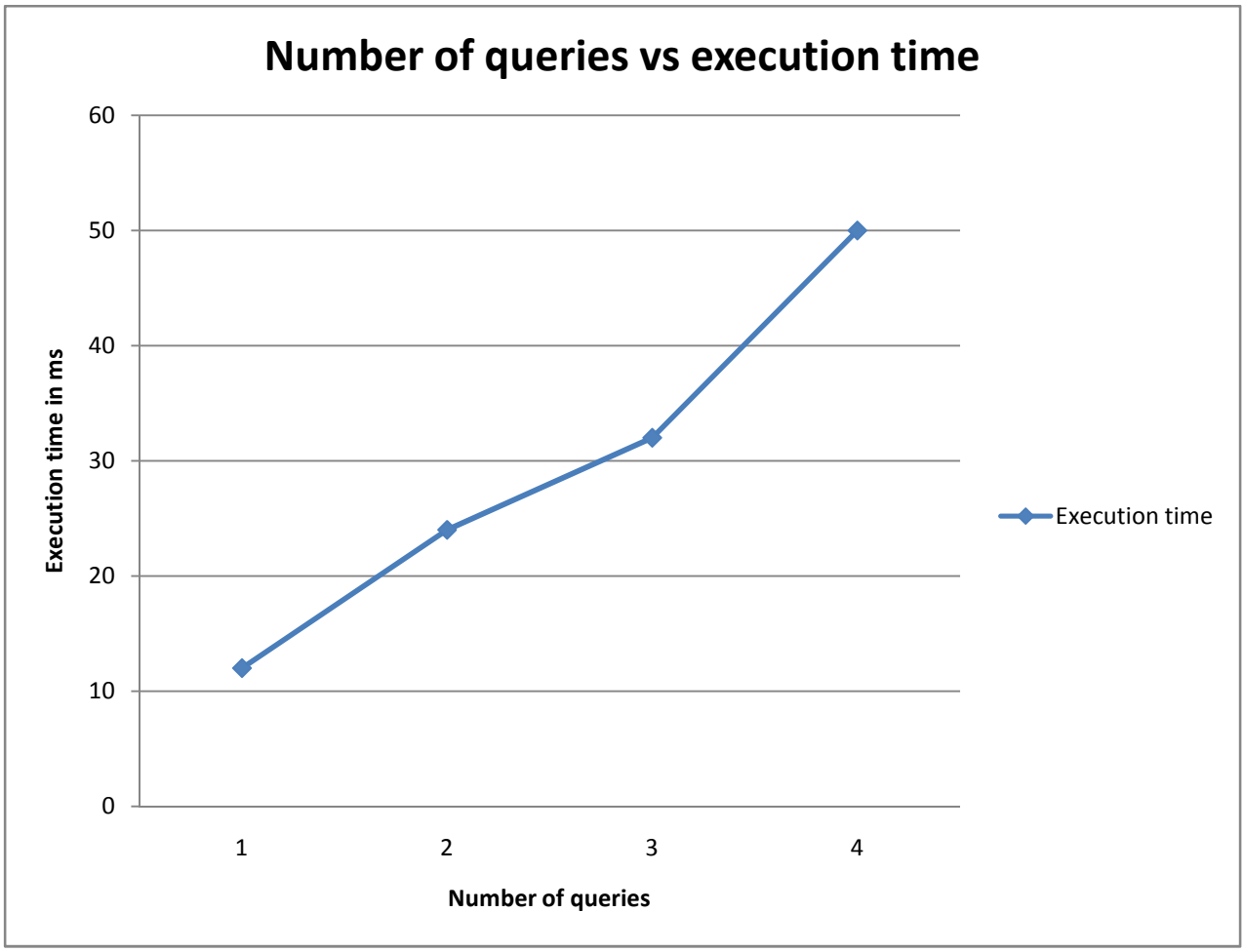


Figure 22: Graph: Number of queries vs execution time with one dependency for RA

Queries with two dependencies each:

Sr No	Number of queries	Average time in ms
1	1	15
2	2	31
3	3	43
4	4	62

Table 6: Execution time for queries with two dependencies for RA

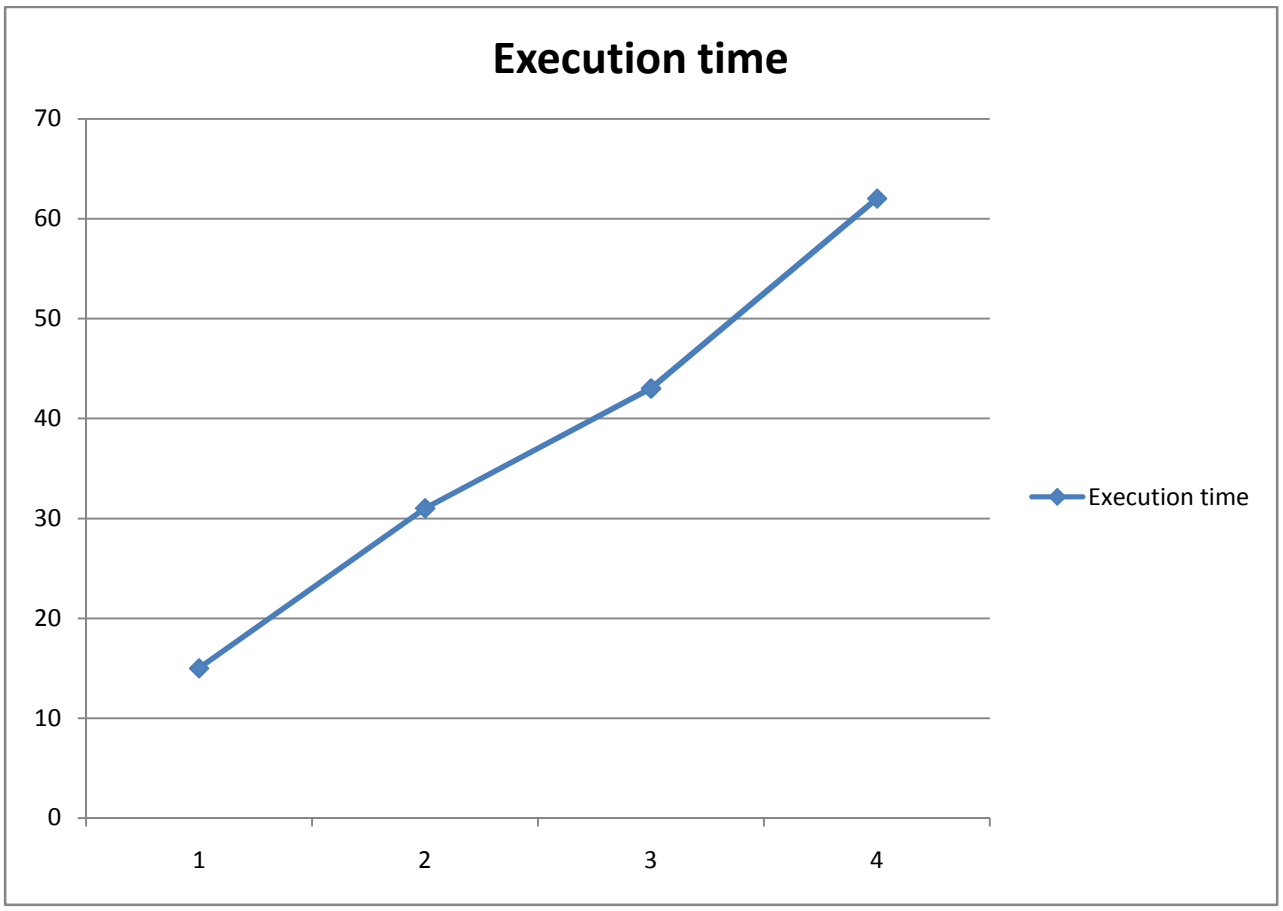


Figure 23: Graph : Number of queries with two dependencies vs execution time for RA

Queries with three dependencies each

Sr No	Number of queries	Average time in ms
1	1	21
2	2	38
3	3	62
4	4	118

Table 7: Execution time of for queries with three dependencies for RA

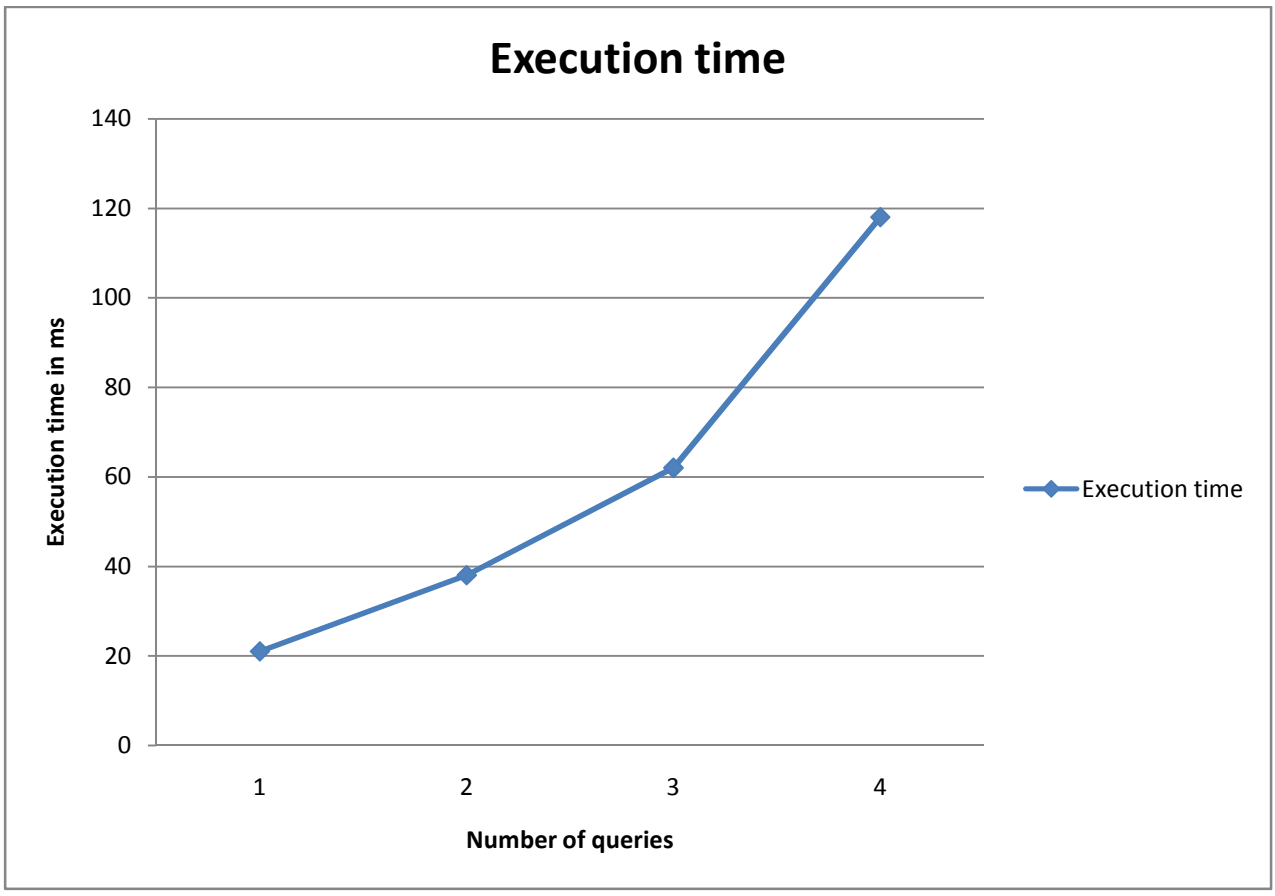


Figure 24: Graph : Number of queries with three dependencies vs execution time for RA

8.0 Conclusions and Future Work

SQL Injection is a common technique that hackers employ to exploit underlying databases in several web and e-commerce applications in the present day. These attacks reshape the SQL query, thus altering the behavior of the program. Although several solutions exist to prevent SQLIA's at the application level, very few solutions exists to prevent them from occurring at the database layer in stored procedures. In this paper, we have studied various SQLIA prevention models and presented a fully automated software implementation for the detection, prevention and reporting of SQLIA's in stored procedures as proposed by Wei, Muthuprasanna et al. The software records the intended SQL query behaviour of all the stored procedures belonging to an application in the form of an SQL-graph, as a one-time offline procedure using static analysis of the stored procedure source code. This graph is then validated against all the different user inputs at runtime to detect all malicious SQL queries, before they are sent for execution. This model helps in catching all of the different types and modes of execution of SQLIAs. We have also provided preliminary evaluation results of the software prototype we developed against the various performance metrics affecting database accesses.

As part of our future work, we plan to work on improving the overall algorithm efficiency, especially that of the runtime analyzer, since it is the one which accounts for the real-time delay in the application response. The static analyzer is run just once for an application (barring any changes in the database) and hence its computational overhead can be ignored for all practical purposes. The runtime analyzer needs to be tested by subjecting it to a variety of practical load situations in a typical client-server scenario to detect performance bottlenecks and remedy them. Further, we also intend to work on extending support of the software to more databases to make it more scalable and comprehensive. Finally, we are also exploring the possibility of implementing this functionality as a middleware to the database engine rather than as an application level library so as to avoid explicit instrumentation of the source code.

9.0 References

1. Hossain Shahriar and Mohammad Zulkernine (Queen's University). *'MUSIC:Mutation-based SQL Injection Vulnerability checking'.* The Eighth International Conference on Quality Software, 2008.

2. Ke Wei, M. Muthuprasanna and Suraj Kothari (Iowa State University). *'Preventing SQL Injection Attacks in Stored Procedures'* .Software engineering conference 2006.

3. Dibyendu Aich (NIT Rourkela). *'Secure Query processing by blocking sql injection'.* M.Tech thesis 2009

4. *Massimo Ficco, Luigi Coppolino and Luigi Romano. 'A Weight-Based Symptom Correlation Approach to SQL Injection Attacks'.*Dependable computing 2009.

5. *Open Web Application Security Project*
   http://www.owasp.org-

6. *Microsoft Developers Network*
   http://msdn.microsoft.com