I.J. Information Engineering and Electronic Business, **2016, 2, 66-75**
Published Online March 2016 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijieeb.2016.02.08

A Parallel-SQLIA Detector for Web Security

Pankaj Kumar
Jawaharlal Nehru University/School of Computer & Systems Sciences, New Delhi, 110067, India
E-mail: pankajkumar.scss.jnu@gmail.com

C.P. Katti
Jawaharlal Nehru University/School of Computer & Systems Sciences, New Delhi, 110067, India
E-mail: cpkatti@yahoo.com

Abstract—An SQL injection attack compromises the interactive web based applications, running database in the backend. The applications provide a form to accept user input and convert it into the SQL statement and fire the same to the database. The attackers change the structure of SQL statement by manipulating user inputs. The existing static and dynamic SQLIA detectors are being used for accurate detection of SQL injection, but it ignores the efficiency of the system. These detectors repeatedly verify the same queries inside the system, which causes unnecessary wastages of system resources. This paper contains the design approach of a parallel algorithm for the detection of SQL injection. The Algorithm uses the concept of Hot Query Bank (HQB) to cooperate with the existing SQLIA detectors (e.g. AMNESIA, SQLGuard, etc) and enhances the system performance. It simply keeps the information of previously verified queries in order to skip the verification process on the next appearance. The system performance has been observed by conducting a series of experiments on multi core processors. The experimental results have shown that parallel-SQLIA detector is 65% more efficient in term of time complexity. Further this design can be implemented in real web application environment; and the design interface can be standardized to cooperate with web application and the SQLIA detectors.

Index Terms—SQL Injection Attack, Hot Query Bank, Web Application, AMNESIA, SQLGuard, Parallel-SQLIA Detectors.

I. Introduction

Web applications are widely used in client/server communication model. It provides a platform for attackers to heck into the database; therefore, their security becomes a major issue. SQL Injection attacks are top ten threats in web application. Every three year, Open Web Application Security Project (OWASP) releases top ten lists of most dangerous security flaws in web applications. Fig.1. shows the ranking of SQL injection. In 2010 onward SQL Injections have become number one attacks in web application [1].

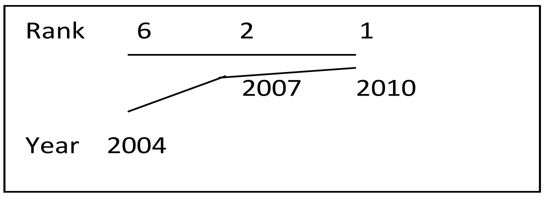

Fig.1. OWASP SQL Injection Ranking

In SQL injection attacks, the attackers inject an input in the SQL query to alter its structure and hence, make access the data in the underlying database. Fig.2. shows a login form with user input.

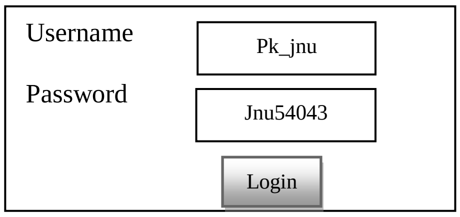
Username
Pk_jnu
Password
Jnu54043
Login

Fig.2. Login form with benign User Input

The above valid input is dynamically generated as an SQL query:

"SELECT * FROM users WHERE username = 'Pk_jnu' AND password = 'jnu5419';"

An attacker might enter malicious input as: Pk_jnu OR '1'= '1' - - in the Username field as shown in Fig.3.

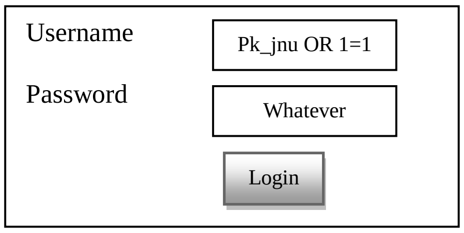
Username
Pk_jnu OR 1=1
Password
Whatever
Login

Fig.3. Login form with Malicious User Input

The above malicious query is dynamically generated as:

"SELECT * FROM users WHERE username = 'Pk_jnu' OR '1'= '1'- - 'AND password = 'whatever';"

The above example only shows a simple SQL Injection scenario. In a real world environment, there are various sophisticated SQL Injections available for web applications.

There are many SQLIA detectors have proposed to prevent the occurrence of SQL Injection attacks. These detectors are either used to detect the vulnerable sources of SQL Injections, or blocks the malicious user inputs. The effectiveness of SQL detectors are measured by the probability of correct judgment. Basically there are two types of error may be possible: false positive, a detector may wrongly identify a legitimate SQL query as an SQLIA; and false negative, a detector may treat a SQLIA query as a legitimate query.

The most of the application having a certain pattern of query, which follows zipf's low. Generally users are more interested for a certain types of data items and query them in frequent manner. As a certain query is more frequently appears and its appearance frequency increase by a given threshold values, these query is treated as a hot query. It would be wastage of time that detectors verify such a hot query.

On the basis of above observation, we have design a parallel-SQLIA detector, which uses the mechanism of Hot Query Bank (HQB) to accelerate the detection process on multi-core processors. HQB is a white listing mechanism, which stored the verified hot queries and intercepts the entire incoming query by inquiring into the verified list of hot query. The query which is not found in the list will be send to the SQLIA detector for verification. The searching speed in HQB is extremely faster than the detection speed of any SQLIA detector. HQB implementation uses efficient hashing techniques. We have measured the performance of system by simulating in various scenarios and it is observe that the system performance can be accelerated up to 65%.

The remainder of this paper is organized as follows. Section II briefly describes the related work about SQLIA detectors. Section III presents Hot Query Bank approach. Section IV illustrates about parallel-SQLIA detector. Section V presents the performance evaluation proposed parallel-SQLIA detector. The final Section concludes the research paper and discuss about future direction.

II. Releted Work

Researchers have suggested many approaches for the protection of web application from SQL injections attacks. Some of these approaches are development based and some are fully automated. This section reviews the proposed defense mechanism and their limitations.

Halfond et al. [2] has classified the SQL injection detection techniques in three categories on the basis of their detection stages: static, dynamic and hybrid.

A. Static

In Static approach, SQL injection attacks are detected at the time of compilation. This approach scans the whole application and then performs heuristics information flow analysis to find bugs in the programs, so that the source code can make bug free or patches for the application can be made. However, it is a time consuming process to deploy the patches for the system. Apart from that, static approach often fails to detect all type of attacks. Most of the time static approaches are unable to capture the actual structure of the query, because the full structure is only available during runtime. In order to sort out the problems associated with static approach many researchers move towards to dynamic approach to analyze the users' inputs and block the malicious content at runtime.

B. Dynamic

Under the dynamic approach the injection attacks are detected at run time. *SQLrand*[6] approach is a dynamic approach, proposed by Boyd and Keromytis. This approach places a proxy server in between web server and SQL server to de-randomize the SQL query received from the clients and send the request to the server and block the vulnerable query during run time. De-randomized query has basically two advantages: Portability and security. It has better performance and imposes maximum of 6.5 millisecond latency overhead. It is considered to be an efficient defense mechanism against injected queries. However, it is a proof of concepts method, so it requires further testing and support by the programmers. SQLCheck: Su and Wasserman [7] implement their algorithm with SQLChecker in a real time environment. This scheme determines the similarity between inputted query and one defined by developer. SQLChecker does not show any false positive or false negative and the computation overhead is also very low. This scheme can be implemented for different applications using different languages. It is an efficient approach, however in one case if an attacker discovers the key then this approach will be compromised. It further requires testing in real web applications. SQLGuard[4]: This is run time technique to eliminate SQL injections. Web based application can easily implements SQLGuard approach against SQL injection. It is a parse tree based approach. The hard-coded portion of the parse tree is supplied by the developer, and the user supplied portion is represented as empty leaf nodes in the parse tree. Users are intended to specify the value of these leaf nodes. A leaf node can simply represent a node in the resulting query that must be the value of a literal.

It has been noted that all kinds of SQL injection alter the construction of SQL query statements intended by the software developer. The structure of an intended query is provided at runtime. After insertion of user supplied input to the input field, the parse tree of this query is compared with the parsed tree of already built intended query and then finds the similarity. The parse tree of injected query and the original query can never be equal and prevent them to execute into the database.

C. Hybrid

It is a combination of static analysis during development and dynamic monitoring at runtime. Hybrid approach is considered to be a most efficient approach that the query statically analyzes to build a query model. During runtime, it checks all the incoming queries with the built query model and then sends it to the database for execution. *AMNESIA* [5] is a tool to detect and prevent SQL injection attacks. It is a hybrid model based approach and it is basically designed to target SQL injection attacks. To analyze the code in web-application, AMNESIA uses static analysis techniques and then automatically builds a model of the query that can be accepted during runtime. It monitors the dynamically generated queries during runtime and checks them for conformity with the statically generated model. When it finds a query that violates the model, it identifies the query as an attack, and prevents it from accessing the database. WebSSARI[8] (Web application Security by Static Analysis and runtime inspection) is a hybrid and taint based approach, that detects error related to input validation by using information flow analysis. This approach can only be able to list out the input query either as a black or white, but it fails to remove the SQL injection vulnerabilities. The static analysis of this approach is used to contain taint flows against the preconditions. The preconditions that have not been met could suggest filters and sanitization functions that can be automatically added to the application to meet the conditions. Sanitized input in WebSSARI system has gone through a predefined set of filters. This approach assumes adequate preconditions for sensitive functions. Which can be accurately expressed using their typing system and the input must pass through certain type of filters. This assumption leads extra burden for many types of subroutines and applications.

III. Hot Query Bank (HQB) Approach

A certain kind of data items in the database are frequently accessed by the users, Hence a particular kind of query constantly appears more frequently and it's frequency increased from a given fixed value, then such query is called as a hot query. This design adds HQB to SQLIA detector to speed up the SQLIA detection process. White listing mechanism is employed to record the verified hot queries. All the incoming queries are intercepted by the HQB by inquiring the recorded query lists during the run time. The query which does not appear in the lists will be considered as a SQL injection and send the same to the detector for verification .The query which is verified as a hot query (i.e. queries found in the list) directly requires database for execution and does not require detector for further verification.

Earlier the test design of HQB was completed and theoretically analyzed its efficiency with existing SQLIA detector in a system. The previous experiments have shown that the SQLIA detectors performance is improved by 45% by the utilization of *HQB approach.* Because of such enhancement and robustness, HQB has promised to provide an additional feature for certain SQLIA detectors for protecting web applications more efficiently. Fig.4. shows a brief flow chart [14] of HQB's functions to counter SQL injection attacks. HQB checks incoming query whether it is an acceptable hot query or not. If it is, then it will be directly send to the database for execution otherwise SQLIA detector will perform further verification. If it examines to be illegitimate then the detector will throw an exception.

The SQL injection detector needs to satisfy the following criteria.

- Eliminate the possibility of attack;
- Minimize the effort required by the programmer;
- Minimize the run-time overhead; and
- Minimize the memory requirement.

Time overhead and memory requirement are the major factor for the performance of SQL Injection attack detection and prevention techniques. Every time a query needs to be verified by the SQL Injection detection technique. It takes a lot of time in verification and memory to store them. Therefore, it requires to finding out some mechanism that can improve the performance of the detection techniques.

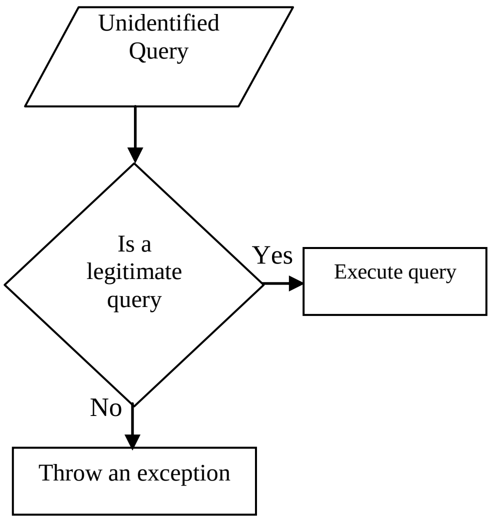

Fig.4. Flow Chart of HQB

IV. Parallel-Sqlia Detector

The design of parallel-SQLIA detector parallelized the HQB approach on multi core processor for further improve the efficiency of the system. It uses Bloom filter [15] as a data structure. Bloom filter (BF) is an efficient data structure, designed to support the membership test for the set of query (i.e. $q \in S$?). Hashing techniques are used for the function of bloom filter. It is an excellent space utilization data structure. BF is an array of size *m* which stores a *m* bits string. Initially, each of the array elements is set to be zero. Let A be the bit string of a BF, and A[i](where $1 \leq i \leq m$) represents as the *i-th* bit of the BF. The BF uses k independent hash functions say h1, h2, h3…h_k; that map the SQL query in the range of{1, 2, 3 ...m}. When a query q_i arrives, then $A[h_j(q_i)]$ is set to 1 for $1 \leq j \leq k$. In order to find membership of any query q_i it checks whether all the bits of $A[h_j(q_i)]$ for $1\le j \le k$ are set to 1 or not. If the value of each of the bits is equals to 1, then it is consider a member of the set *S*. Otherwise, not a member. A BF may generate a BF error [14], due to the hash collision. Bloom filter may suggest that a query q_i is a member of the set, while it is not. Assume that each array position is equally likely to select by a hash function. If array contains m bits and k is the number of hash functions, then the probability that a certain bit is not set to 1 by a certain hash function will be

$$(1-1/m)$$

The probability that it is not set to 1 by any one of the hash function is:

$$(1-1/m)^k$$

If n elements are inserted then the probability of a certain bit is still 0 is:

$$(1-1/m)^{kn}$$

Thus, the probability of certain bit 1 is:

$$[1-(1-1/m)^{kn}]$$

Hash function compute each of the *k* array positions is 1 with a probability $[1-(1-1/m)^{kn}]$. Then the probability that all of them being 1 is given as:

$$f=[1-(1-1/m)^{kn}]^k \approx [1-e^{-kn/m}]^k \tag{1}$$

For a given value of *m* and *n* the value of *k* is:

$$K=m/n\ln 2$$

$$2^{-k} \approx 0.6185^{m/n} \tag{2}$$

A. Sliding Window

Let *S* be a query stream and *W* be the sliding window size. Each query *q* in the query stream is designated with a timestamp *q.t* to indicate its arrival time. A query *q* is a valid query only if $q\cdot t \in (t_{now}-W, t_{now})$ .where t_{now} is the current time.

Let the occurrence frequency of a query *q* in the sliding window is *f(q)* and N denote the sum of all frequencies in the sliding window, that is:

$$N=\sum \forall_{q\cdot t\in(t_{now}-W,t_{now})} f(q) \tag{3}$$

Let *s* be the support parameter, and the value of $s \in (0,1)$.

If $f(q) \ge sN$, then q is a hot query. For example, suppose there are four different queries *q1*, *q2*, *q3* and *q4* with each arrival time shown in the Fig.5. Where t_{now} =10 and *W*=6. Note that query $q_4\cdot t \notin (t_{now}-W, t_{now})$ = (4, 10). Let the occurrence frequency of query *q1*, *q2* and *q3* are 3, 1 and 1 respectively in the sliding window. Thus $N=f(q1)+f(q2)+f(q3)$=3+1+1=5. Assume support parameter *s*=0.3. Therefore, only query *q1* is qualified as a hot query in this example. Because $f(q_1)$=3≥sN=0.3×5=1.5. And the remaining queries are treated as a cold query and require further verification in their next appearances.

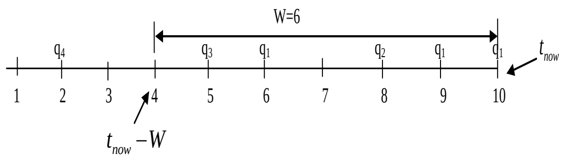

Fig.5. An Example of Sliding Window

B. The Parameters

The Parallel-SQLIA detector has multiple bloom filters [14] (e.g. HQB.size is number of BFs). The BFs are arranged according to increasing order of last access time (LAT). *BF[i]*is *i-th* BF of SQLIA detector, *BF[i].n* is the number of queries, stored in *BF[i]*, and *BF[i].LAT* is the last access time of *i-th* bloom filter, *BF[i].th* be the threshold of each BF's size. The new BF will be appended to SQLIA detector if $BF[i].n \ge th$.

The relationship among the bloom filter error probability *f*, bloom filter size *m*, number of hash function *k* and threshold value *th* is as follows:

$$k=\lfloor -\log_2 f \rfloor \tag{4}$$

$$th=m/k(\ln 2) \tag{5}$$

Let f_b be their error probability with which two BF collaborates to acquire a global BF error probability *f*.

f =1-[(There is no BF error in BF [1]) ∩ (There is no BF error in BF [2])].

$$f= 1-(1-f_b)(1-f_b) =1-(1-f_b) 2$$

$$f_b=1-\sqrt{1-f} \tag{6}$$

Where k_b denotes the number of hash function and th_b represents the number of queries that a *BF* can store.

$$k_b=\lfloor -\log_2 f_b \rfloor = \lfloor -\log_2(1-\sqrt{1-f}) \rfloor \tag{7}$$

$$th_b = m / k_b (\ln 2) \tag{8}$$

In general, if the detector has υ bloom filters then

$$k_b = \lfloor -\log_2 f_b \rfloor = \lfloor -\log(1 - \sqrt[\upsilon]{1-f}) \rfloor \tag{9}$$

Table 1. The Relationship Among υ, f_b, th_b, K_b and N

υ	f_b	th_b	k_b	N
1	0.05	1732	4	1732
2	0.025320566	1386	5	2772
3	0.016953428	1386	5	4158
4	0.012741455	1155	6	4620
5	0.010206218	1155	6	5775
6	0.008512445	1155	6	6930
7	0.007300832	990	7	6930
8	0.006391151	990	7	7920
9	0.005683045	990	7	8910
10	0.005116197	990	7	9900

C. Design and Implementation

In this technique *HQB* coordinated with *SQLGuard*[4] and *AMNESIA*[5] detector. Moreover the algorithm is a parallel algorithm to run on multi-core processors for further enhances the performance of the detector. Therefore the performance of *SQLGuard* and *AMNESIA* has been improved. This section presents system architecture; flow chart, data structure and parallel algorithm design. All the algorithms are implemented on open MP or P-Thread Library machine. It requires gcc compiler for the execution. The experiments have been performed on Ubuntu machine with Intel Dual Core CPU (2.0 GHz) and 2 GB RAM.

a. System Architecture

HQB [14] acts as a bridge between web application and database as shown in Fig.6. It intercepts and analyzes queries from web application to a database. When a query is suspected as an SQL injection then it will through an exception note over the web application; else it is considered to be legal and will be transmitted to the database for execution.

b. Data Structure

Hot Query Bank provides a method which is similar to *hCount* [14] to get hot queries. It provides three additional capabilities (i) it proposed a mechanism which prevents bloom filter errors, as it might mistreat a certain cold queries as a hot one and vice versa and thus threaten the system security. (ii) the dynamic data set can be handled by *HQB* and (iii) Query repository is used to store recent hot queries.

Parallel-SQLIA detector contains a set of bloom filters. At the beginning, detector will contain only one bloom filter and the additional bloom filter will be added as per requirements. The incoming queries are kept into the Bloom filter until it frequency reaches to a fixed threshold values. The algorithm keeps the track of the last access time (*LAT*) of each bloom filter. *LAT* denote the last update time of a bloom filter. *LAT* can be used to find whether a BF expires or not. Moreover, a BF will expire if *LAT* of the BF is smaller than t_{now} -*W*. The expired BFs have been dropped from HQB for efficient utilization of memory. Moreover the detector maintains a data structure called Query Repository (*QR*) [14] to store the contents of hot query as shown in Fig.7.

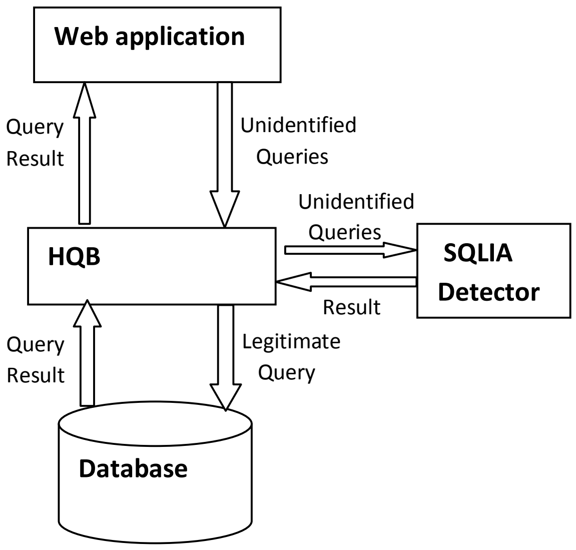

Fig.6. Architecture of SQLIA Detector

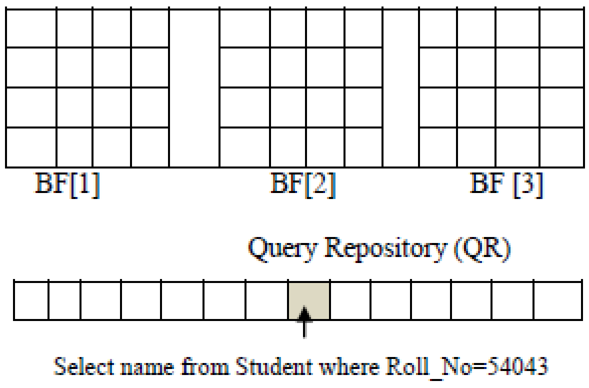

Fig.7. The Data Structure

c. Flow Chart and Algorithms

The Algorithm1 verifies a query *q* whether it is legitimate or not. It performs the three major tasks(i)updates the frequency of a query *q* (ii)check whether query *q* is hot or not and (iii) verify whether *q* is a legitimate query or not. The flow chart is shown in Fig.8. At first the algorithm checks whether '*q*' exists in Query Repository (*QR*) [14] or not. If it does, '*q*' must be a hot and legal query, then its time of occurrence will be updated, and a true value will be returned.

If it does not exist in *QR* then '*q*' may not be become hot, but it might be a legal query, *or* it may be a *SQL injection*. In both cases, the algorithm will send *q* to the

SQLIA detector for verification. If *q* is legal, then the variable *query_legitimate* (query legitimate is a Boolean variable) will be set true. According to the status of the variable *query_legitimate,* there can be two possible cases:

- ***Query_Legitimate is false:*** *'*q*' is a* SQL injecion, *and detector will come out with an exception.*
- ***Query_Legitimate is true:*** *'q' is not hot but legal one.*

Algorithm 2 and 3 are act as SQL injection detectors. Algorithm2 illustrates the functionality of SQLGuard detector and Algorithm 3 of AMNESIA. If the detectors verify the inputted query '*q*' as a legitimate, then the frequency of legitimate query 'q' will be recorded and finally the task will be completed by calling an insertion function, implemented in Algorithm 5. Subsequently, parallel-SQLIA detector calls *Hot- Query* function (i.e., in Algorithm4) to verify whether *q* is a hot query or not. If it is so then *q* may have just become a hot query from a cold one after its continuous occurrence. Therefore, the SQLIA detector will store q in *QR* and return true.

d. The Time Complexity of Algorithm1

The time complexity of *Algorithm1* mainly falls on four parts: (i)the cost for searching a query in *Query repository*(ii) Cost of *SQLIA* detector for verifying a query(iii) Parallel_insert function to insert a query into bloom filter and (iv)the Parallel*_isHotQuery()* function to check the hotness of a query. The cost for searching a query in *QR* is *O(1)* because *QR* is implemented using a Hash table. Let *R* be the probability that a query '*q'* is not legitimate, then the time complexity for sending query to *SQLIA* detector will be $O(R\times T_{detector})$, where $T_{detector}$ is time taken by *SQLIA* detector to verify a query. Since we are assuming that there are *p* number of processor and all are working simultaneously, so the time complexity of Parallel*_insert ()* function and Parallel*_isHotQuery()* function is *O(1)* and *O(1)* respectively. Therefore the time complexity of Algorithm1 becomes $O(R\times T_{detector})$ (i.e. only detector time).

e. Determination of recently hot query

The frequency of a query '*q'* is estimated by *Algorithm4* and it also decides if '*q'* is a recent hot query or not. How many time that '*q'* has appeared in the sliding window are recorded by a variable total_freq (i.e., at Line 4 of Algorithm4), and *N* denotes the sum of all queries' frequency inside the sliding window. The minimum value of associated counter is used to estimate *q's* frequency, and then accumulate the estimated frequency to total_freq. Check *q's* frequency with estimated frequency, and returns the result either true or false to *Algorithm1*. Workload of Algorithm4 mainly falls between Line 2 and 4, where *HQB* needs to go through all *BF*. Here we have been considering a multiprocessor system having *p* number of processor, and each BF needs to do the hashing up to *k* times. Therefore, its time complexity is O(k) and the number of processor required is *p=O(HQB.size).*

f. Legitimate Query insertion

Algorithm5 illustrates the detail process of Query insertion. At first *HQB* checks whether the size of the last

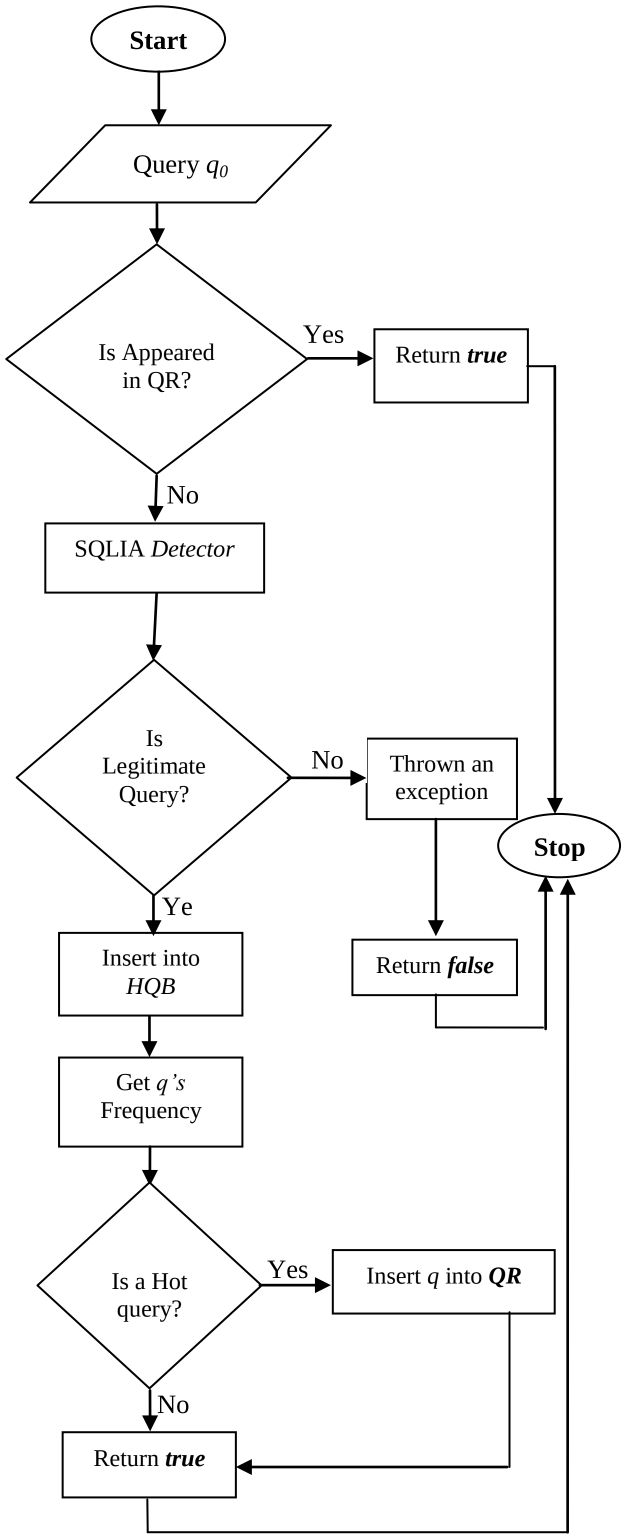

Fig.8. Flow Chart

BF exceeds the threshold value (*th*) or not. If it exceeds from *th*, then a new *BF* will be appended to the *HQB. HQB* then inserts '*q'* into the last *BF* and increases *n* of the last *BF* by one. *HQB* also fixes the current time (i.e.,

t_{now}) to *BF.LAT*. The insertion of a query ‘*q’* into a bloom filter requires computation of k hash function which is done by *p* number of processor simultaneously, so the time complexity of *Algorithm5* is *O (1)*.

Algorithm1

MIMD_SM: Multiple instruction multiple data shared memory Architecture machine
P: No. of processor

```
Parallel_SQLIA_Detector(MIMD_SM)
{
  1. Global_query_appeared←false
  2. ∀ P_i Where 0≤i≤p-1do
     i.   Local_query_appread←QR[h_i(q)]
     ii.  Lock(global_query_appread)
     iii. Global_query_appread←Global_query_appread+
          local_query_appread
     iv.  End for
  3. if(Global_query_appread=true)then
     i.  q.t←t_now
     ii. return true
  4. else
     i.   query_legitimate←send query "q" to SQLIA Detector
          for verification(e.g. AMNESIA, SQLGuard)
     ii.  if(query_legitimate=true)then
          a. q.t=t_now
          b. Parallel_insert(q)
          c. query_hot←Parallel_isHotQuery(q)
          d. if(query_hot=true)then
                 Insert 'q' to QR[h_k(q)]by processor p_k and
               return true
     iii. else
           Thrown exception and return true
  5. Exit
}
```

Algorithm2

SQLGuard_Detector(q): Verify whether a Query ‘q’ is legitimate or not.
Input: A query ‘q’ will be passed to the SQLGuard detector
Output: Return true if Query ‘q’ is legitimate hot otherwise false.

```
1. Make the Parse tree of  the inputed Query
2. Match the parse tree of  inputed query and already built
   parse tree of that query.
3. If both the parse tree are matched then the SQL query q is
   legitimate.
And return true.
4. Otherwise not legitimate and return False.
5. Exit
```

Algorithm3

AMNESIA_Detector (): Verify whether a set of Query within the application are legitimate or not.
Input: An application containinmg SQL query, will be passed to the *AMNESIA* detector
Output: Return *true* ***if*** Query ‘*q’* is legitimate hot otherwise *false*.

```
1. Scan the entire application and identify the Hot Spot (i.e.
   SQL Queries) to the underlying database.
2. Identified query passed to a Non Deterministic Automata in
   which transition level consists of SQL tokens, delimitors
   and space for SQL String value.
3. If the Automata reaches to the final state then SQL query is
   a legitimate query.And return true.
4. Otherwise return false.
5. Exit.
```

Algorithm 4

```
Parallel_isHotQuery(q)
{
  1. #pragma omp parallel
     i.   Total_freq←0
     ii.  N←0
     iii. Min_freq← ∞
  2. ∀ p_i where BF[i] ∈ HQB do
     i.  If(BF[i].LAT< t_now)then Continue
     ii. N←N+BF.n
  3. ∀ p_i where i←1 to k do
     i.  If(BF.A[i][h_i(q)]=0)then
         a. Min_frq←0
         b. Break
     ii. If(BF.A[i][h_i(q)]≤Min_freq)then
         a. Min_freq←BF.A[i][h_i(q)]
  4. Total_freq← Total_freq+Min_freq
  5. Return(Total_freq≥sN)
}
```

Algorithm 5

```
Parallel_Insert(q)
{
  1. If(BF[HQB.Size]≥th)then
     i.  Add a new BF to HQB
     ii. HQB.Size=HQB.Size+1
  2. BF_current←BF[HQB.Size]
  3. ∀ P_i where i←1 to k do
     i.  BF_current.A[i][h_i(q)]←BF_current.A[i][h_i(q)]+1
     ii. End for
  4. BF_current .n←BF_current .n+1
  5. BF_current .LAT←t_now
}
```

g. Maintenance of HQB

The legitimate hot queries [14] are store in Query Repository (QR). The size of QR must be equal to the number of all hot and legitimate queries. The design of *QR* should be as such, it provides the following advantages (i) fewer numbers of hot queries requires lesser amount of memory. (ii) fast search speed due to the small size of *QR* and (iii) high occurrence frequency of hot legitimate queries.

Notice that if Query Repository provides the above advantages, then the verification of '*q'* as a legal query is possible by simply checking *QR*. Therefore, query 'q' is no more required to send to the *SQLIA* detector for verification, and thus improve the overall performance.

V. Performance Evaluation

A series of simulation experiment has been performed in order to evaluate the performance of *Parallel-SQLIA* detectors under different parameter settings. The detector is basically combination of *HQB* [14] and *SQLGuard* [4] called *HQBGuard* detector and *HQB* and *AMNESIA* [5] called *HQBAMNESIA* detector. *SQLGuard* uses a parse tree approach and *AMNESIA* is a model based approach. AMNESIA uses finite automata for the construction of SQL model. The performance of the system has been compared among the parallel-SQLIA (*HQBGuard, HQBAMNESIA*) detectors, *SQLGuard* and *AMNESIA* detectors. Here the "performance" refers to the total execution time.

A. Experimental setup

The parameters used in the experiment are summarized in a *Table 2*. There are 10 test samples generated in order to perform experiment. Each of them contains 10,000 legitimate queries with 0%, 0.01%, 0.1%, 1%, 5%, 10% and 20% malicious queries taken from the literature of *SQL injection attacks*. One query is to be selected from a file containing 10,000 queries every time. The experiment has been terminated after examined the verifications of 100,000 queries. The simulator selects the queries based on two key factors: the skew coefficient of a Zipf distribution (ϴ) [14] and the ratio of malicious queries (r).

r= Number of malicious queries / Total number of queries.

For example, let **r** =10% and *Ntotal* = 100,000 then malicious query will be 10,000(approx). The simulator flips a biased coin with the probability **r** for heads to select a query from the total query. The simulator selects a malicious query randomly and uniformly, if the outcome of the coin is head. Otherwise, a legal query is picked by the simulator based upon the Zipf distribution (ϴ).

Let $\Pr(q_i)$ be the probability of selecting i-th query out of 10,000 legal queries.

$$\Pr(q_i) = \frac{\left(1/i \right)^{\theta}}{\sum_{i=1}^{10000} \left(1/i \right)^{\theta}} \tag{10}$$

Where, θ is the skew coefficient and *i* is the Rank of the Query based on frequency. Highest query's frequency has given *rank 1*, second highest *2*, and so on. The frequency of hot query is directly proportional to the skew coefficient.

$$f(q_i) \propto \theta \tag{11}$$

Therefore, if the frequency of a query increases, then the Algorithm will take lesser time to recognize as a Hot Query.

$$f(q_i) \propto 1/Time \tag{12}$$

Table 2. Parameter Setting

Parameter	Value
Number of total queries(Ntotal)	100,000
Percentage of malicious Queries(r)	0%, 0.01%, 0.1%, 1%, 5%, 10%, 15%, 20%
The Skew coefficient of Zipf distribution(ϴ)	0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.0
Support Parameter(s)	0.001, 0.005, 0.01, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4
The size of the sliding window(W)	500, 1000, 1500, 2000, 2500, 3000
Bloom filter error probability(f)	0.05
The size of Bloom filter(m)	10,000
The number of hash function(kb)	6
Update time interval(Tupdate)	1000, 2000, 3000, 4000, 5000

If the number of hot queries increase then the skew coefficient will also increase and the distribution of $Pr(q_i)$ will become more skew. Here we have considered the sliding window size (*W*) as 500, 1000, 1500, 2000, 2500 and 3000. It is assumed that a query arrives one at a time. A query is submitted only after the completion of previous query. Here we have consider the sliding window can contains maximum of 3000 queries (i.e., *N* =3000).

Let Bloom filter error probability (*f*) is 0.05, and the Bloom filter size (*m*) is 10,000 (i.e.10, 000 queries can accommodate in one Bloom filter). Let the number of hash functions (K_b) is 6.

B. *Effects of Update Time Interval (T_{update})*

The update interval ranges from 1000 to 5000 and the execution time is measured at every interval. Fig.9. shows the total execution time of the detector on varying T_{update}. The most interesting thing is that the T_{update} does not have any insignificant impact on the performance of parallel-SQLIA detector. *SQLGuard*[4] needs to generate two sets of parse tree for each input SQL statement, and then compare with each other. Whereas the *AMNESIA* [5] requires to generate a finite automata for the given SQL statement and check whether it reaches to final state or not, which tends to cause much computation cost.

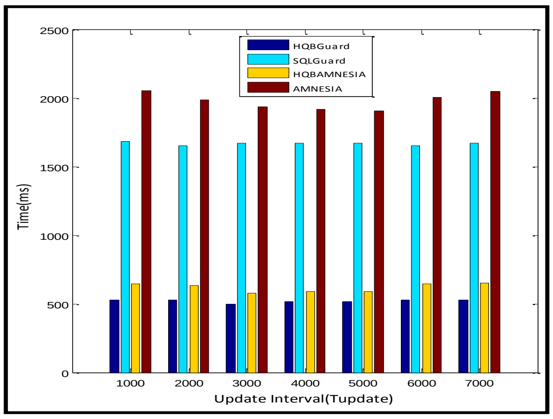

Fig.9. Effects of Update Time Intervals

C. *Effects of the Ratio of Vulnerable Queries (r)*

In this case ,there are two interesting facts can be observed by the experimental results (i) the parallel-SQLIA detector always performs better than those of *SQLGuard*[4] and *AMNESIA*[5]. (ii) as the value of *r* increases (i.e. if malicious query ratio is more in the system), then the performance of Parallel-SQLIA detector and the detector alone (i.e. SQLGuard, AMNESIA) would accordingly decreases. When a malicious query appears, Parallel-SQLIA detector sends it to the normal detector for further verification. Likewise, the SQLGuard spends more time to parse a malicious query and AMNESIA to construct a finite automaton for malicious query; which results in a poor performance. Fig.10. shows the execution time of SQLGuard, AMNESIA and Parallel-SQLIA detector under various ***r*** values.

D. *Effects of Support Parameter (s)*

Fig.11. shows the execution time of *SQLGuard*, *AMNESIA* and *Parallel-SQLIA* detector under various support parameter values. It illustrates the impact of support parameter (*s*) over execution time. A smaller value of support parameter(*s*) means that higher number of hot query exists in the system. It is notice that *parallel* detector need not necessary send hot queries to the *SQLIA* detector, thus it saves the time for their verifications. This experiment shows that *Parallel-SQLIA* detector performs better with a smaller value of support parameter (*s*).

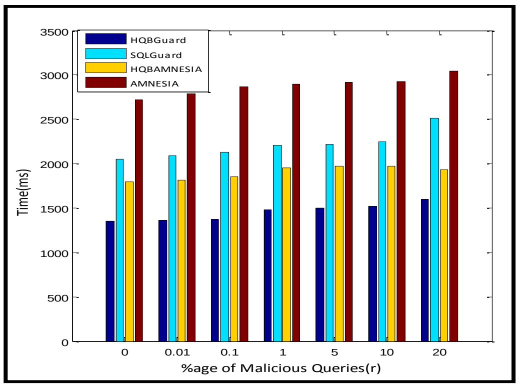

Fig.10. Effects of the Ratio of the Vulnerable Queries

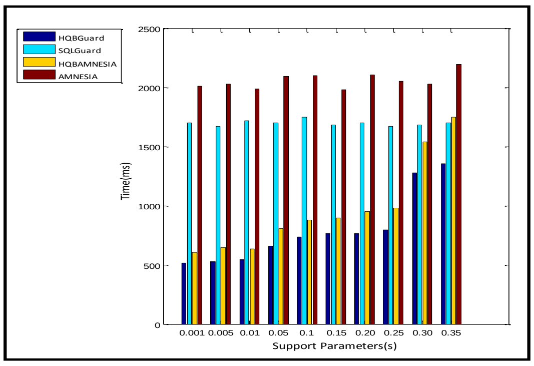

Fig.11. Effects of Support Parameters

E. *Effects of Size of Sliding Windows (w)*

An experimental result shows that if the sliding window (W) size increases then the execution time of parallel-SQLIA would slightly increases. Parallel-SQLIA detector performs better than *SQLGuard*[4] and *AMNESIA*[5] SQLIA detector under all size of windows (W). If the size of sliding window is large then *HQB* requires more number of bloom filters for storing the query information, which in turn requires more space and incurs more computation cost. On the other hand the existing SQLIA detectors does not have sliding window concept so it does not considered the factor of sliding window, thus the sliding window size will not affect its performance. Fig.12. shows the graph of experimental results on varying window size.

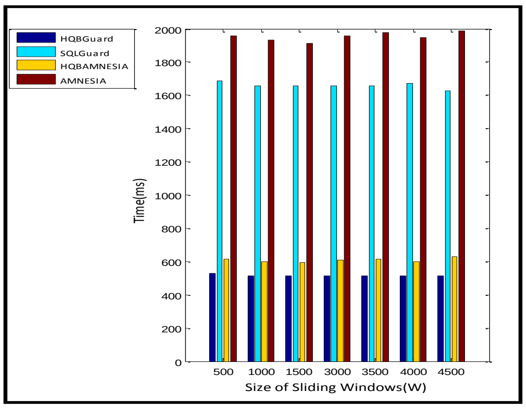

Fig.12. Effects of Size of Sliding Window

VI. Conclusion and Future Work

The Parallel-SQLIA detector can detect and prevent all types of SQL injection vulnerabilities as well enhances the overall system performance. Literature review suggested most of the existing SQLIA detectors that repeatedly verify the same query inside the system. Such repetitions cause unnecessary wastage of time and system resources. Parallel-SQLIA detector simply keeps the information of hot queries and skips the repeated verification; therefore improve the system performance in terms of execution time. We have performed a series of experiments to measure the respective performance. The experimental results have shown that parallel -SQLIA detector is 65% more efficient in term of time complexity, regardless of any kind of detectors is being used.

This research work also provides some possible future direction. Since the parallel detector has been demonstrated through simulation study; further it can be tested in a real web application. The standard interface could be designed, in which HQB can cooperate with any other SQLIA detectors.

References

[1] OWASP Top Ten Project. Owasp top 10 application security risks, 2010.

[2] W.G. Halfond,J. Viegas, and A. Orso, "A classification of SQL-injection attacks and countermeasures," In Proc.of the IEEE Intl .Symp. on Secure Software Engineering, Mar 2006.

[3] C. A. Mackay (Jan 2005), SQL Injection Attacks and Some Tips on How to Prevent Them [Online]. Available: http://www.codeproject.com/cs/database/SQlInjectionAttacks.asp.

[4] G. Buehrer, B. W.Weide, and P. A. G. Sivilotti, "Using parse tree validation to prevent SQL injection attacks," In Proc. of the 5th intl. Workshop on Software engineering and middleware, SEM '05, New York, NY, USA, pp.106–113, 2005.

[5] W. G. Halfond and A. Orso, "AMNESIA: Analysis and monitoring for neutralizing SQL-injection attacks," In Proc. of the IEEE and ACM Intel. Conf. on Automated Software Engineering (ASE 2005), Long Beach, CA, USA, Nov 2005.

[6] S.W. Boyd and A.D. Keromytis, "SQLrand: Preventing SQL injection attacks," In Proc. of the 2nd Applied Cryptography and Network Security (ACNS'04) Conference, pp. 292-302, Jun 2004.

[7] Z. Su, and G. Wassermann, "The essence of command injection attacks in web application," In ACM Symposium on Principles of Programming Languages (POPL'2006), Jan 2006.

[8] Y.W. Huang, F. Yu, C. Hang, C.H. Tsai, D.T. Lee, and S.Y Kuo, "Securing web application code by static analysis and runtime protection," In Proc. of the 13th Intl. Conf. on World Wide Web, New York, pp. 40-52, 2004.

[9] M. Martin, B. Livshits and M. S. Lam, "Finding application error and security flaws using PQL: A program query language," In Proc. of the 20th annual ACM SIGPLAN conference on Object oriented programming systems, languages and applications (OOPSLA 2005), pp. 365-383, 2005.

[10] R.A. McClure, and I.H. Kruger, "SQL DOM: Compile time checking of dynamic SQL statements," In Proc. of the 27th Intl. Conf. on Software Engineering (ICSE 2005),nos. 15-21, pp. 88-96, May 2005.

[11] P. Bisht, P. Madhusudan, and V.N. Venkatakrishnan, "CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks," ACM Transactions on Information and System Security, vol. 13, no. 2, 2010.

[12] S. Ali, S.K. Shahzad, and H. Javed, "SQLIPA: An authentication mechanism against SQL injection," European Journal of Scientific Research, vol. 38, no. 4, pp. 604-621, 2009.

[13] M. Junjin, "An approach for SQL injection vulnerability detection," In Proc. of the 6th Intl. Conf. on Information Technology: New Generations 2009 (ITNG'09), nos. 27-29, pp. 1411-1414, Apr 2009.

[14] Y.C. Chang,M.C. Wu, Y.C. Chen, W.K. Chang, "A hot query bank approach to improve detection performance against SQL injection attacks," Computers& Security, vol. 31, no. 2, pp. 233-248, Mar 2012.

[15] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo, "The dynamic bloom filters, "IEEE Transaction on Knowledge and Data Engineering, vol. 22, no. 1, pp.120-133, Jan 2010.

Authors' Profiles

Pankaj Kumar is PhD scholar in School of Computer and Systems sciences, Jawaharlal Nehru University (JNU), New Delhi, India. He received B.E degree in Information Technology from Sant Longowal Institute of Engineering and Technology, Sangrur, Punjab, India in 2011 and degree of M.Tech in Computer Science and Technology from JNU in 2014. His research area is Computer Network Security and Cryptography.

C.P. Katti is Professor and Dean in School of Computer and Systems Sciences, Jawaharlal Nehru University (JNU), New Delhi, India. He received degree of M.S. in Applied Mathematics from University of Missouri, Columbia, MO., USA in 1976 and awarded PhD in Scientific Computation/ Numerical Analysis from IIT Delhi in 1981. His area of research is parallel processing and scientific computing. He published more than 30 papers in international journals of repute.