Detection of SQL Injection Attacks using Hidden Markov Models

Ajit Kumar Sahoo*, and Khushboo Agarwal†
*Department of CSE, Silicon Institute of Technology, Bhubaneswar,
†Department of CSE, Silicon Institute of Technology, Bhubaneswar

Abstract—**In this Internet age, web applications have become an integral part of our lives, but security & privacy of our sensitive data has become a big concern. Over last several years, SQL Injection has been the most prevalent form of attack on web databases. Much research has been done in this area, but most of the approaches in the literature have high computational overhead or difficult to deploy in practical scenarios. In this project, we intend to find out a solution to identify SQL injected queries at the database firewall level, i.e., between the web application and the database server. We aim to build a system that can examine run-time queries issued by a web application for execution and allow only benign queries to pass through to the database server. Any SQL query is just a string, containing SQL keywords, values, identifiers, delimiters etc. Therefore, detecting injected queries can be considered as a pattern recognition problem. Hidden Markov Model (HMM) is a technique which has been successfully used in many pattern recognition areas such as speech recognition, handwriting recognition etc. Based on our preliminary study, a database firewall consisting of HMMs can be trained to identify SQL injected queries at run-time. We will focus primarily on web applications developed with PHP & MySQL. However, because the system is targeted to work as a database firewall, it will be portable and technology/platform independent. We will be using PHP, MySQL, and MATLAB etc. for this project.**

1. Introduction

In recent years, widespread adoption of the internet has resulted in to rapid advancement in information technologies. The internet is used by the general population for the information, in a way that allows the information owners quick access while blocking break-in attempts from unauthorized users. Every year about 50% of databases experience at least one purposes such as financial transactions, educational endeavors, and countless other activities. The use of the internet for accomplishing important tasks, such as transferring a balance from a bank account, always comes with a security risk. Today's websites strive to keep their users' data confidential and after years of doing secure business online, these companies have become experts in information security. The database systems behind these secure websites store non-critical data along with sensitive security breach. The loss of revenue associated with such breaches has been estimated to be over four million dollars. Additionally, recent research by the "Imperva Application Defense Centre" concluded that at least 92% of web applications are susceptible to "malicious attack".

Among various types of security threats, web applications are exposed to, SQL Injection attack is predominantly used against web databases. The Open Web Application Security Project (OWASP) ranks it on top among the Top-10 security threats. According to TrustWave 2012 Global Security Report, SQL injection was the number one attack method for four consecutive years. Nowadays, attackers use sophisticated Botnets which automatically discover vulnerable web pages from search engines like Google and launch mass SQL injection attacks from distributed sources. About 97% of data breaches across the world occur due to SQL injection alone.

Although recently there has been a great deal of attention to the problem of SQL injection vulnerabilities, many proposed solutions fail to address the full scope of the problem. There are many types of SQL Injection Attacks and countless variations on these basic types. Researchers and practitioners are often unaware of the myriad of different techniques that can be used to perform SQL Injection Attacks. Therefore, most of the solutions proposed detect or prevent only a subset of the possible SQL Injection Attacks. Here we propose a novel method for detection of SQL injection using Hidden Markov Model.

2. Related Work

Researchers have proposed a wide range of techniques to address the problem of SQL injection. These techniques range from development best practices to fully automated frameworks for detecting and preventing SQL Injection Attacks. In this section, we review these proposed techniques and summarize the advantages and disadvantages associated with each technique. Various techniques have been proposed for the confrontation of the threat of SQL injection attacks. In this section, the characteristics of the best known techniques [1] are briefly discussed and their principal weaknesses are highlighted.

A. STATIC ANALYSIS

Wassermann and Su propose a static analysis framework to filter user inputs. According to them, their approach has some limitations concerning implementation-related issues, such as the way it handles some operators. Additionally, this approach is limited to discover only tautology-based attacks, i.e. attacks that always result in true or false SQL statements. Another static analysis approach has been proposed by Livshits and Lam. In this work, vulnerability patterns are described in a program query language called PQL. Static analysis [2] is applied to find potential violations matching a vulnerability pattern.

The main limitation of the method is that it cannot detect SQL injection attacks patterns that are not known beforehand, and explicitly described in the specifications. In Huang et al., preconditions are specified for all sensitive PHP functions and user input is checked against these preconditions. Like all static analysis approaches, his technique does not provide an automated mechanism for detection and prevention of SQL injection attacks. This is the most important reason why all the approaches of this subsection generate a significant number of false negatives.

B. *DYNAMIC ANALYSIS*

AMNESIA [3] uses a model-based approach to detect illegal queries before their execution into the database. In its static part, the technique uses program analysis to automatically build a model of the legitimate queries that could be generated by the application. In its dynamic part, the technique uses runtime monitoring to inspect the dynamically-generated queries and check them against the statically-built model. A primary assumption regarding the applications which the method targets is that the application developer creates queries by combining hard- coded strings and variables using operations such as concatenation, appending and insertion. The key insights behind the development of the technique are that the information needed to predict the possible structure of the queries generated by a web application is contained within the application's code, and an SQLIA, by injecting additional SQL statements into a query, would violate that structure. Therefore, our technique first uses static program analysis to analyze the application code and automatically build a model of the legitimate queries that could be generated by the application. Then, at runtime, the technique monitors all dynamically-generated queries and checks them for compliance with the statically-generated model. Queries that violate the model are classified as illegal, prevented from executing on the database, and reported to the application developers and administrators. The main drawback of AMNESIA is that it requires the modification of the web application's source code for the successful collaboration with the security monitor-officer. Also it involves a number of steps using different tools.

SQLGuard is based on comparing, at run time, the parse tree of the SQL statement before the inclusion of the user input with that resulting from parse tree of the SQL statement after the inclusion of the user input. A secret key is used for wrapping the user input, so if an attacker compromises this key, SQLGuard is difficult to prevent an attack. The overhead to database query costs is about 3msec (the characteristics of their server were very similar to ours: a 733MHz Windows 2000 Server machine with 256MB RAM) . Another drawback of their method is that, similarly to AMNESIA, it requires the modification of the application's scripts.

SQLCheck is a similar approach. It adds a key at the beginning and at the end of each user's input. At runtime, the "augmented" queries that are not in a valid syntactic form are considered attacks. The detection ability of the approach depends on the strength of the key also.

In SQLrand [4] , the SQL standard keywords are manipulated by appending a random integer to them that an attacker cannot easily guess. Therefore, any malicious user attempting an SQL injection attack would be thwarted. We create randomized instances of the SQL query language, by randomizing the template query inside the CGI script and the database parser. To allow for easy retrofitting of our solution to existing systems, we introduce a de-randomizing proxy, which converts randomized queries to proper SQL queries for the database. Code injected by the rogue client evaluates to undefined keywords and expressions. When this is the outcome, then standard keywords (e.g., "or") lose their significance, and attacks are frustrated before they can even commence. The performance overhead of our approach is minimal, adding up to 6.5 ms to query processing time. This technique uses a key to randomize SQL queries, so if an attacker compromises this key, SQLRand is difficult to prevent an attack. Moreover, the approach imposes a significant overhead in terms of infrastructure because it requires the integration of a special proxy in the web-application infrastructure.

3. BACKGROUND

The increasing number of Web-based applications that are deployed worldwide makes protection a key topic in computer security. Unfortunately, the large number of new attacks that appear every day makes impossible to have signature-based systems always updated to the most recent attacks.

We intend to find out a solution to identify SQL injected queries at the database firewall level, i.e., between the web application and the database server. We aim to build a model [5] trained using Hidden Markov Model that can examine run-time queries issued by a web application for execution and allow only benign queries to pass through to the database server.

A. *SQL Injection Attack*

An SQL Injection Attack (SQLIA) [6] occurs due to unverified/unsanitized input vulnerability i.e. when an attacker attempts to change the logic, semantics or syntax, and behavior of a legitimate dynamically generated SQL statement by inserting additional SQL keywords and/or operators into the statement using URL query string or HTML form input box values, usually with a malicious intent. SQL injection vulnerability on a web page exists if user inputs or query string parameters are used to construct dynamic SQL queries without properly validating them. According to OWASP, this is the most common flaw found in web applications.

To explain the basic mechanism of SQL injection, we take the classic example of a login form, commonly seen on many websites. A typical PHP code snippet that validates login credentials against the values stored in databases is as below:

```
<?php
$sql = "SELECT * FROM tbl_users WHERE
username ='".$_POST['user_name']"'AND
password =' "$_POST['user_pass']"';
$result = mysql_query($query,$connection);
```

?>

If the user enters ‘john’ as the user name and ‘mypass’ as the password, the resulting SQL query that will be executed against the database is:

```
SELECT * FROM tbl_users WHERE username =
'john' AND password = 'mypass';
```

However, if an attacker enters “abcd’ OR 1 = 1– ” as the user name and “xyz” as the password, then the resulting query after substituting these values would be:

```
SELECT * FROM tbl_users WHERE username =
'abcd' OR 1 = 1--'AND  password= 'xyz'
```

It may be noted that the single quote after “abcd” in the input serves as the string terminator for the username value while the added clause “OR 1 = 1” always evaluates to TRUE. The double-dash at the end of the inputted username, which is the comment character in Transact-SQL, effectively comments out the rest of the query. When executed, the query returns all records from the tbl_users table with the record pointer at the first row of the result set. The attacker therefore successfully logs in as the first user in the table. The fundamental fact here is the ability of an attacker to change the syntax and semantics of the query by carefully inserting special characters and SQL keywords into the input fields which drastically alter the behavior of the dynamically generated query originally intended by the programmer, thereby resulting in an unauthorized access into the protected area of the website.

B. Hidden Markov Models

A Hidden Markov Model (HMM) [7] is a statistical Markov model in which the system being modeled is assumed to be a Markov process with unobserved (hidden) states. A HMM can be presented as the simplest dynamic Bayesian network. The mathematics behind the HMM was developed by Baum. In simpler Markov models (like a Markov chain), the state is directly visible to the observer, and therefore the state transition probabilities are the only parameters. In a hidden Markov model, the state is not directly visible, but output, dependent on the state, is visible. Each state has a probability distribution over the possible output tokens. Therefore the sequence of tokens generated by an HMM gives some information about the sequence of states. Hidden Markov models are especially known for their application in temporal pattern recognition such as speech, handwriting, gesture recognition, part-of-speech tagging, musical score following, partial discharges and bioinformatics.
A HMM Model [8] is specified by:
S - The set of states (n is the number of states)

$$S = \{s_1, s_2, \ldots, s_n\}$$

π- The prior probabilities as the probabilities of s_i being the first state of the sequence.
A - The transition probabilities as the probabilities to go from state i to state j.

$$A_{ij} = P(Q_{n+1} = s_j | Q_j = s_i)$$

B - The emission probabilities characterize the likelihood of a certain observation $x_n \in v_1, \ldots, v_k$:, if the model is in state s_i.

$$B_{ik} = P(x_n = v_k | Q_n = s_i)$$

Mathematically, HMM can be represented as

$$\lambda = (A, B, \pi)$$

Where,
A = state transition probabilities
B = observation probability matrix
π = initial state distribution
The matrices π, A and B are row stochastic, meaning that each element is a probability and the elements of each row sum to 1, that is, each row is a probability distribution. A HMM allowing for transitions from any emitting state to any other emitting state is called an ergodic HMM. The other extreme, a HMM where the transitions only go from one state to itself or to a unique follower is called a left-right HMM. There are three fundamental problems that we can solve using HMMs. Here, we briefly describe these three problems.

1) Problem 1 EVALUATION: Given the model λ = (A,B,π) and a sequence of observations O,find P(O |λ). Here, we want to determine the likelihood of the observed sequence O, given the model.

2) Problem 2 DETECTION: Given λ = (A, B, π) and an observation sequence O, find an optimal state sequence for the underlying Markov process. In other words, we want to uncover the hidden part of the Hidden Markov Model.

3) Problem 3 TRAINING: Given an observation sequence O and the dimensions N and M, find the model λ = (A,B,π) that maximizes the probability of O. This can be viewed as training a model to best fit the observed data.
As we want to detect SQL injection, we wish to approach the problem by first training our HMM (Problem 3) in offline mode, then checking a dynamic query at run-time by evaluating against the trained hmm parameters (Problem 1).
SQL queries are simple tokens of characters. As HMM has been proved very effective in pattern recognition problem, we expect HMM to be equally effective in our problem.

4. PROPOSED APPROACH

A. Query Transformation Scheme

The transformation scheme [9] normalizes an SQL query into a sentence-like form, and facilitates application of similarity and hmm training. The extended scheme uses only capital A-Z for all tokens and space character as token separator. All symbols and special characters are also transformed into words except the underscore (_) character, because it is frequently used in MySQL system databases, system tables and functions. Splitting such tokens at the underscore character would result in over-token formation and negatively affect the similarity values.
Each step of query transformation is a find-and-replace operation, done by appropriately using the preg_replace() and str_replace() built-in functions available in PHP.

For example, preg_replace("/0x[0-9a-f]+/i", "HEX", $query) converts hexadecimal numbers in $query into HEX in a case-insensitive manner.
All symbols and special characters are transformed in as per the scheme given in the table. The last two steps convert the transformed query to uppercase and reduce multiple spaces to single spaces. To visualize how the transformation scheme works, consider the following SQL queries, intentionally written in mixed-case to show the effect of transformation.

```
sEleCt * fRoM products wHeRe price > 10.00
aNd discount<8 SeLeCt email FrOm customers
WhErE fname LIKE 'john%'
```

By applying the transformation scheme [10] , the above queries are transformed into the following form respectively:

SELECT STAR FROM USRTBL WHERE USRCOL GT DEC AND USRCOL LT INT SELECT USRCOL FROM USRTBL WHERE USRCOL LIKE SQUT STR PRCNT SQUT

Consider an injected query generated due to a cleverly crafted attack to bypass detection:

```
SELECT * FROM products WHERE prod_id = 24
OR 'ABC' = CoNcAt(cHaR(0x28 + 25),
cHaR(0x42), cHAr(80 - 0x0d))
```

This query contains a number of symbols and operators, which is not suitable for application of HMM training. The transformation scheme converts it completely into text form as:

SELECT STAR FROM USRTBL WHERE USRCOL EQ
INTOR SQUT STR SQUT EQ CONCAT LPRN CHAR
LPRN HEX PLUS INT RPRN CMMA CHAR LPRN
HEX RPRN CMMA CHAR LPRN INT MINUS HEX
RPRN RPRN

Any SQL query, irrespective of its complexity, is thus transformed into a series of words separated by spaces like a sentence in English. The structural form of the query is correctly maintained by the transformation scheme.

B. Detection Strategy

Design of SQL Injection Attacks is strategically guided by two interesting observations which we were unable to and mentioned anywhere in the literature. The first observation surfaced while looking for an answer to "where do SQL injections most commonly occur in an injected query?" Looking at the general coding practices for developing web applications, we and that, dynamic SQL queries are usually constructed in two parts:
(1) A static part hard coded by the programmer, and
(2) A dynamic part produced by concatenation of SQL keywords, delimiters and received input values.

Since the main objective of a dynamic query is to fetch different set of records depending on the input, the parameter values must be used in the WHERE clause to specify the selection criteria. This compulsive programming need as well as the de facto practice of using input values in the WHERE clause part of dynamic queries resolves that SQL injections happen after the WHERE keyword, almost in all cases.

In fact, SQL injection before the WHERE keyword is extremely rare [10] – not a single instance was found in over 16,500 queries containing SQL injection attacks we examined for this study. The second interesting observation stems from the basic intention behind an SQL injection attack – stealing sensitive information from the back-end database, such as credit card numbers. By carefully formulating the injection code, the attacker tries to get the intended data displayed on the web page itself, which is the only method for data breach through SQL injection. Data is fetched by SELECT queries in SQL, which implies that unless the injectable parameter is used to construct a dynamic SELECT query that delivers data displayed on the web page, it is not useful for data breach.

It points to another assertion that SQL injections mostly occur through dynamically generated SELECT queries. This leads to the strategic inference that, it is sufficient to examine the part of an SQL query after the WHERE keyword in order to detect the presence of any injection attack, which significantly narrows down the scope of processing.

Another plausible strategy may be to intercept only SELECT queries for examination; however, this may enable the attacker to cause damage to the data, if not breach. By considering the part of a query after the WHERE keyword, we automatically include UPDATE and DELETE queries in the investigation, because they also support a WHERE clause by SQL syntax.

It is interesting to note that:
1) The victim dynamic query must be a SELECT query fetching data for display on the web page, and
2) The stored value containing injected code must be used in its WHERE clause.

By detecting injection in the WHERE clause, the second order attack would also be rendered ineffective. Therefore, the strategy to examine only the portion of queries after the WHERE keyword is correct and sufficient for detecting SQL injection attacks, including second order injections. After this query transformation scheme, number of queries can be reduced strategically.

C. Cosine Dissimilarity Strategy

But we couldn't train one hmm for all these queries. For effective results, we needed to train different HMMs for different types of queries. So, to cluster our query set into different groups, we employed the clustering algorithm.

In a wide sense, a similarity index measures [11] the degree to which a pair of objects is alike. Similarity measure is the just the complement of distance measure.

Let s_1 =$(x_{11}x_{21}\ldots x_{L11})$ and s_2 = $(x_{12}x_{22}\ldots x_{L22})$ be strings defined over the alphabet Σ, and let $|s_i| = L_i$ denote the string's length. A similarity index, S(s_1, s_2), measures the degree to which a pair of objects are alike; the larger its value, the closer or more alike we think the patterns are. Conversely, dissimilarity coefficients, d (s_1, s_2), assess the degree to which patterns differ, smaller values meaning closer or higher resemblance. Distances, differences, reciprocal of similarities, all constitute examples of dissimilarity measures. Given the inverse relationship between similarity and dissimilarity, a simple way to transform similarity into dissimilarity, in the situation of bounded ranges, is:

$$d(s_1, s_2) = S_{max} - S(s_1, s_2) \tag{1}$$

There are different similarity measures we can consider for our problem. We have considered cosine similarity suitable for our problem. It quantifies correlation between vectors A and B as cosine of the angle between them in n-dimensional space. In this approach each query is treated as a vector having the dimension as 685 as we have 685 different tokens in the query transformation model. Now if A and B are two different vectors then we can compute the cosine similarity is expressed as follows

$$CosineSimilarity = \frac{|\overrightarrow{A}.\overrightarrow{B}|}{|\overrightarrow{A}|.|\overrightarrow{B}|} \tag{2}$$

Also the dissimilarity can be calculated as follows:

$$CosineDissimilarity = 1 - CosineSimilarity \tag{3}$$

D. Clustering Using K-Medoid

Clustering [12] is the task of grouping a set of objects in such a way that objects in the same group (cluster) are more similar (in some sense or another) to each other than to those in other groups (clusters). It is a main task of exploratory data mining, and a common technique for statistical data analysis, used in many fields, including machine learning, pattern recognition m image analysis, information retrieval and bioinformatics. The main advantage of clustering is that interesting patterns and structures can be found directly form very large data-sets with little or none of the background knowledge.

Mathematically, consider a data set D consisting of N sequences of symbols (or strings), D = $(s_1 s_2 \dots s_n)$, defined over the alphabet Σ, where $s_i = (x_1^i x_2^i \dots x_{L_i}^i)$ is a sequence of length L_i. The problem of sequence clustering is defined as follows: Given the sample patterns in D, discover from the data a "natural" grouping of the sequences into K clusters, K being in general unknown.

We can group the similar queries so that the number of HMMs required to train will be less. As we have the dissimilarity matrix in our hand so we can go for the k-Medoid clustering algorithm.

The K-means algorithm is sensitive to outliers since an object with an extremely large value may substantially distort the distribution of data. Instead of taking the mean value of the objects in a cluster as a reference point, a Medoid can be used, which is the most centrally located object in a cluster. Thus the partitioning method can still be performed based on the principle of minimizing the sum of the dissimilarities between each object and its corresponding reference point. This forms the basis of the K-Medoids method.

The basic strategy of K- Mediods clustering algorithms is to find k clusters in n objects by first arbitrarily finding a representative object (Medoids) for each cluster. Remaining object is clustered with the most similar object. K-Medoids method uses representative objects as reference points instead of mean value of the objects in each cluster.

The algorithm[9] takes the input parameter k, the number of clusters to be partitioned among a set of n objects. A typical K-Mediods algorithm for partitioning based on Medoid or central objects is given below. It outputs a set of k clusters that minimizes the sum of the dissimilarities of all the objects to their nearest medoid.

Algorithm 1: K-Medoids Algorithm

```
Require: Input K(Number of Cluster), D(Data Set)
1: Arbitrarily choose k objects in D as the initial representative objects;
2: repeat
3:     Assign each remaining object to the cluster with the nearest medoid;
4:     Randomly select a non medoid object $O_{random}$ ;
5:     Compute the total points S of swap point $O_j$ with $O_{random}$;
6:     if S < 0 then
7:         swap $O_j$ with $O_{random}$ to form the new set;
8:     end if
9: until No Change in new set of medoids
```

1) Advantages: It is Simple to understand and implement.Also it is fast and convergent in a finite number of steps and usually less sensitive to outliers than k-means.It allows using general dissimilarities of objects.

2) Disadvantages: Different initial sets of Medoids can lead to different final clustering. It is thus advisable to run the procedure several times with different initial sets of Medoids. Moreover, the resulting clustering depends on the units of measurement. If the variables are of different nature or are very different with respect to their magnitude, then it is advisable to standardize them.

5. PROPOSED APPROACH

HMM Training: In training phase 90% queries of each cluster is used for the training of each HMM(rest 10% is kept for training purpose) and a value of Hidden Markov model parameters λ = (A,B,π) are obtained.

Run Time Detection: When a query is received at run-time, after normalization, it is passed through each of the HMMs and evaluated against the calculated set of hmm parameters and then the likelihood is checked. The hmm which gives the maximum likelihood decides whether the query is generated or injected.

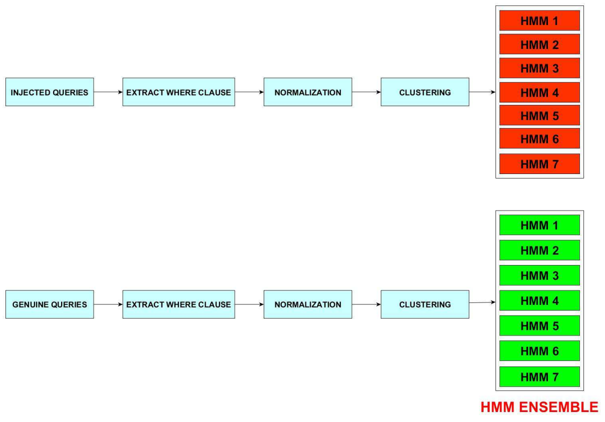

Figure 1: HMM Training

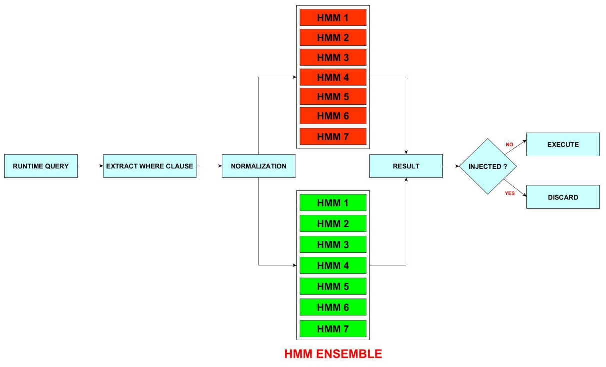

Figure 2: Run Time Detection

6. EXPERIMENATAL EVALUATION

In this phase the rest 10% queries of each cluster formed in earlier stage is used for detection using the evaluate procedure of the Hidden Markov Models. And the results are recorded along with the error percentage.

		PREDICTED	
		GENUINE	INJECTED
ACTUAL	GENUINE	TN = 50	FP = 3
	INJECTED	FN = 4	TP = 496

Table I: Confusion Matrix

Accuracy	0.987314
True Positive Rate (Recall)	0.992
False Positive Rate	0.056603
True Negative Rate	0.943396
False Negative Rate	0.008
Precision	0.993987

Table II: Result Matrix

Our results were obtained with **98.73%** accuracy and **99.39%** precision.

7. PROPOSED APPROACH

We calculated the running time of each of our process for both genuine and injected queries. The performance overhead was then calculated. The time overhead was more for injected queries compared to genuine queries as the injected queries are usually longer. Moreover, the time overhead is more for training process but it can be ignored as the HMM training is done just once and that too in offline mode. The time overhead for run-time detection is very less i.e. 5ms for genuine and 13ms for injected queries which shows the effectiveness of our system, incurring least overhead on our system.

8. CONCLUSION

In this project, we have first used query normalization scheme for transforming the queries to simple sentence like form. We adopted the strategy to examine only the WHERE clause part of run-time queries and ignore INSERT queries. This approach minimizes the size of the legitimate query repository. The system required an initial set of injected as well as genuine queries, these were collected from few automated tools and some web applications. We then used K-Medoids algorithm with Cosine Dissimilarity approach to cluster our queries.

The clusters were used to train an HMM ensemble, consisting of HMMs (one for each cluster). The HMM parameters obtained are used to check any query at run-time for deciding whether the query has to be executed or discarded.

The results are very encouraging and confirm effectiveness of our approach. The accuracy of the system is calculated to be 98.73%. Performance overhead of the system is almost imperceptible over Internet. The system acts as a database firewall and is able to protect multiple web applications hosted on a shared server, which is an advantage over existing methods. The approach can also be ported to other web application development platforms without requiring major 525.

REFERENCES

[1] K. Kemalis and T. Tzouramanis, “Sql-ids: a specification-based approach for sql-injection detection,” in *Proceedings of the 2008 ACM symposium on Applied computing*. ACM, 2008, pp. 2153–2158.
[2] D. A. Kindy and A.-S. K. Pathan, “A survey on sql injection: Vulnerabilities, attacks, and prevention techniques,” 2011.
[3] W. G. Halfond and A. Orso, “Amnesia: analysis and monitoring for neutralizing sql-injection attacks,” in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 2005, pp. 174–183.
[4] S. W. Boyd and A. D. Keromytis, “Sqlrand: Preventing sql injection attacks,” in *Applied Cryptography and Network Security*. Springer, 2004, pp. 292–302.
[5] D. Ariu, R. Tronci, and G. Giacinto, “Hmmpayl: An intrusion detection system based on hidden markov models,” *computers & security*, vol. 30, no. 4, pp. 221–241, 2011.
[6] W. Halfond, J. Viegas, and A. Orso, “A classification of sql-injection attacks and countermeasures,” in *Proceedings of the IEEE International Symposium on Secure Software Engineering*, vol. 1. IEEE, 2006, pp. 13–15.
[7] M. Stamp, “A revealing introduction to hidden markov models,” *Department of Computer Science San Jose State University*, 2004.

[8] B. Resch *et al.*, "Hidden markov models a tutorial for the course computational intelligence," 2004.

[9] D. Kar and S. Panigrahi, "Prevention of sql injection attack using query transformation and hashing," in *Advance Computing Conference (IACC), 2013 IEEE 3rd International*. IEEE, 2013, pp. 1317–1323.

[10] D. Kar, S. Panigrahi, and S. Sundararajan, "Sqlidds: Sql injection detection using query transformation and document similarity," in *Distributed Computing and Internet Technology*. Springer, 2015, pp. 377–390.

[11] D. K. Roy and L. K. Sharma, "Genetic k-means clustering algorithm for mixed numeric and categorical data sets," 2010.

[12] T. Velmurugan and T. Santhanam, "Computational complexity between k-means and k-medoids clustering algorithms for normal and uniform distributions of data points," *Journal of computer science*, vol. 6, no. 3, p. 363, 2010.