Accepted Manuscript

Title: SQLiGoT: Detecting SQL Injection Attacks using Graph of Tokens and SVM

Author: Debabrata Kar, Suvasini Panigrahi, Srikanth Sundararajan

PII: S0167-4048(16)30045-1
DOI: http://dx.doi.org/doi: 10.1016/j.cose.2016.04.005
Reference: COSE 998

To appear in: *Computers & Security*

Received date: 24-9-2015
Revised date: 26-3-2016
Accepted date: 16-4-2016

Please cite this article as: Debabrata Kar, Suvasini Panigrahi, Srikanth Sundararajan, SQLiGoT: Detecting SQL Injection Attacks using Graph of Tokens and SVM, *Computers & Security* (2016), http://dx.doi.org/doi: 10.1016/j.cose.2016.04.005.

SQLiGoT: Detecting SQL Injection Attacks using Graph of Tokens and SVM

Debabrata Kar[a,1,*], Suvasini Panigrahi[b,2], Srikanth Sundararajan[c,3]

[a]*Silicon Institute of Technology, Bhubaneswar, India*

[b]*VSS University of Technology, Burla, Sambalpur, India*

[c]*Indian Institute of Technology, Bhubaneswar, India*

*Corresponding author

Email address: debabrata.kar@silicon.ac.in (Debabrata Kar), spanigrahi_cse@vssut.ac.in (Suvasini Panigrahi), sundararajan.srikanth@gmail.com (Srikanth Sundararajan)

[1]Department of Computer Science and Engineering

[2]Department of Computer Science & Engineering & IT

[3]Currently at Helion Advisors, Bangalore, India

Biographical Sketch of Authors

Debabrata Kar received B.Sc. Engg from NIT, Rourkela and M.Tech in Computer Science from Utkal University, Bhubaneswar. He has over 12 years of experience in web application development and is a PhD candidate at School of Computer Engineering, KIIT University, Bhubaneswar. His research interests include cyber security and database security.

Suvasini Panigrahi received her B.Tech followed by M.Tech in Computer Science Engineering from Utkal University, Bhubaneswar, and PhD from IIT Kharagpur. Her research interests include intrusion detection, database security, fraud detection, wireless sensor networks and computer graphics.

Srikanth Sundararajan obtained B.Tech from IIT Madras, and MS/PhD in Computer and Information Sciences from University of Illinois, Urbana Champaign. He has 25+ years of international experience in the software product and services space. He has worked at HP, Informix, HCL, and top positions in Cognizant and Persistent. He is also a visiting faculty of Computer Science at IIT Bhubaneswar and has taught at several US Universities. He has several publications and is the holder of patents, jointly with HP and BEA.

Abstract

SQL injection attacks have been predominant on web databases since last 15 years. Exploiting input validation flaws, attackers inject SQL code through the front-end of websites and steal data from the back-end databases. Detection of SQL injection attacks has been a challenging problem due to extreme heterogeneity of the attack vectors. In this paper, we present a novel approach to detect injection attacks by modeling SQL queries as graph of tokens and using the centrality measure of nodes to train a Support Vector Machine (SVM). We explore different methods of creating token graphs and propose alternative designs of the system comprising of single and multiple SVMs. The system is designed to work at the database firewall layer and can protect multiple web applications in a shared hosting scenario. Though we focus primarily on web applications developed with PHP and MySQL, the approach can be easily ported to other platforms. The experimental results demonstrate that this technique can effectively identify malicious SQL queries with negligible performance overhead.

Keywords: sql injection attack, sql injection detection, query token graph, graph of tokens, node centrality, database firewall, support vector machine

1. Introduction

Access to internet on hand held devices like smart phones, tablets etc., have made web content ubiquitously available. With our dependence on web applications, the amount of sensitive and personally identifiable information stored in back-end databases has also scaled up. Web databases are highly lucrative targets for hackers, therefore securing them is a prime concern. Unfortunately, most web applications are developed without due attention towards security (Scholte et al., 2012a). Many websites are built with off-the-shelf open source packages and third party plugins without verifying the security aspects of the code. As a result, web applications often get deployed with multiple vulnerabilities which an attacker can exploit. According to TrustWave (2015), 98% of web applications have one or more security vulnerabilities.

SQL injection attack (SQLIA) is a simple and well understood technique of inserting an SQL query segment usually through GET or POST parameters submitted to a web application. It occurs when untrusted user input data is used to construct dynamic SQL queries without proper

validation. Exploiting this commonly found flaw, a hacker can extract, modify or erase the content in the back-end database. SQLIA can also occur through cookies and other HTTP headers. The classic example of "`OR 1 = 1`" injection, known as a *tautological* attack, is usually given to introduce SQLIA. There are however several types of injection attacks depending on the technique used and attack intention (Halfond et al., 2006). In spite of growing awareness among web developers, from 2013 to 2014, SQLIA has increased by 10%, and number of vulnerable web applications has increased by 9% (TrustWave, 2013, 2015). Nowadays, attackers use sophisticated and automated tools or Botnets (Maciejak and Lovet, 2009) which can automatically discover vulnerable web pages from search engines like Google (Long et al., 2011) and launch attacks en masse.

To protect against SQL injection, Intrusion Detection Systems (IDS) like Snort and Web Application Firewalls (WAF) like Apache ModSecurity are used, but they are still penetrable. Most WAFs rely on regular expression based filters created from known attack signatures, and require lot of expert configuration. Signature-based detection can be circumvented using different techniques (Maor and Shulman, 2004; Carettoni and Paola, 2009; Dahse, 2010; Quartini and Rondini, 2011). Tools like WAFw00f (Gauci and Henrique, 2014) and WAFFle (Schmitt and Schinzel, 2012) can identify and fingerprint the WAF deployed. Since SQL injection vectors can be formed in numerous ways, attacks can be specifically devised to bypass the perimeter security. For example, consider the following SQL injection vectors:

```
OR 'ABCDEF' = 'aBcDeF'
OR 419320 = 0x84B22 - b'11110010100101010'
OR PI() > LOG10(EXP(2))
OR 'abc' < cOnCaT('D', 'E', 'F')
OR 'DEF' > CoNcAt(ChAr(0x41), cHaR(0x42), chAr(0x43))
OR 'XYZ' = sUbStRiNg(CoNcAt('a', 'BcX', 'Y', 'ZdEf'), 4, 3)
```

All of these are tautological expressions. Such heterogeneity makes it extremely difficult to construct regular expressions or rule-based filters. Large number of filters also become a performance bottleneck. In fact, all WAFs can be bypassed with cleverly crafted injection vectors (Lupták, 2011).

In spite of excellent research to prevent and detect SQLIA, many approaches fail to address the complete scope of the problem or have limitations preventing their application in a practical environment. In this paper, we present a novel technique to detect SQLIA by modeling an SQL query as a graph of tokens and using the centrality measure of nodes to identify the

malicious ones by a trained SVM. The concept of graph of tokens (also referred to as terms or words) are typically used in Natural Language Processing (NLP), information retrieval, keyword extraction, sentiment analysis in social media and several other domains. We first normalize an SQL query into a sequence of tokens and generate a graph capturing the interaction between the tokens. The centrality measure of nodes is used to train an SVM classifier. The technique was implemented in a prototype named SQLiGoT (SQL injection Detection using Graph of Tokens) and was validated experimentally on five sample web applications with large set of legitimate accesses and injection attacks. The contributions of this paper are as follows:

1. A method to normalize SQL queries into a sequence of tokens while preserving the syntactical structure is described.
2. A technique to generate a weighted graph that models the normalized query as an interaction network between the tokens is introduced.
3. It is demonstrated that the centrality measure of the tokens (nodes) can be used to train an SVM-based classifier system.
4. Both directed and undirected token graphs are explored with two different weighting schemes and the results are compared.
5. A generic SQLIA detection system is described and alternative designs consisting of single and multiple SVMs are proposed.

The rest of the paper is organized as follows. Section 2 presents a brief review of research existing in the literature. Section 3 states the SQLIA detection problem and lays down our motivation behind this study. Section 4 describes the core components of our approach. The generic architecture of the proposed system is introduced in Section 5 and alternative designs based on single and multiple SVMs are discussed. Experimental evaluation is presented in Section 6 along with assessment of performance overhead. Quoting some related works in Section 7, we conclude the paper in Section 8 with a note on future directions of research.

2. State of the art

SQLIA has long gained attention of researchers and has a rich literature. We briefly review some existing approaches based on five critical layers in the end-to-end web application architecture (Fig. 1), at which the solutions have been proposed. These are: (1) web client, (2) web application firewall, (3) web application, (4) database firewall, and (5) database server. We also quote some solutions which target or utilize information from multiple layers.

2.1. *Web client*

Shahriar et al. (2013) proposed to detect SQLIA at the web client (browser) by sending shadow SQL queries in hidden form fields and validating the dynamic query generated by user inputs through JavaScript functions before submitting the form. The method is flawed in several ways: it does not consider URL parameters, requires instrumenting the web application, hidden fields can be seen in the source HTML, JavaScript can be disabled on the browser, and an attacker would prefer to use a tool instead. To our knowledge, this is the only attempt to detect SQLIA at the client side.

2.2. *Web application firewall (WAF)*

Approaches at this layer focus on examining the HTTP request and finding anomalies for SQLIA detection. Moosa (2010) proposed a neural network based firewall trained with character distributions of legitimate and malicious HTTP requests. A separate instance of their system is required for each website on shared hosts, which is a major practical drawback. Alserhani et al. (2011) considered SQLIA as a multi-stage activity and proposed correlating isolated alerts generated by Snort IDS. Their approach requires an IDS deployed on the web server. Scholte et al. (2012b) developed IPAAS that learns data types of GET and POST parameters during training and enforces validation at runtime. The system cannot prevent attacks through parameters which accept free text. Makiou et al. (2014) suggested a hybrid system using Bayesian classifier and pattern matching against a rule set. The system is optimized for low false negatives but yields high false positives. While occasional false negatives may not be concerning, false positives may obstruct smooth operation of a web application. WAF based approaches require more computation for processing the entire HTTP request and are generally prone to higher false alarms.

2.3. *Web application*

Majority of contributions focus at the web application level, categorizable into: defensive coding, vulnerability testing, and prevention based approaches. Defensive coding approaches require programmers to write the application code in a specific manner. Mcclure and Kruger (2005) designed SQL-DOM, a set of classes strongly typed to the database schema, which are used to generate safe SQL statements. The approach has very high time complexity. Also, any change to database schema requires regeneration and recompilation of the classes. Thomas et al. (2009) and Bisht et al. (2010c) proposed methods to automatically generate prepared statements

in existing application code. The original source code must be maintained, so that the prepared statements can be regenerated upon modification of the application. Vulnerability testing based approaches rely on testing web applications to discover possible injection hotspots so as to fix them before production release. Benedikt et al. (2002) designed VeriWeb and Jovanovic et al. (2006) developed Pixy to automatically test and discover SQL injection vulnerabilities. Shin et al. (2006), Wassermann et al. (2008), and Ruse et al. (2010) proposed automatic generation of test inputs & cases to help application testers. Bisht et al. (2010a) designed NoTamper, a black-box testing method for detecting server-side vulnerabilities. Effectiveness of these approaches is limited by their ability to discover all possible security issues. Prevention based approaches consist of preparing a model of queries and/or the application's behavior during normal-use and using the model to detect anomalies at runtime. Halfond and Orso (2005) designed AMNESIA combining static analysis of source code and runtime monitoring. Buehrer et al. (2005) proposed SQLGuard which compares parse tree of queries before and after user-input inclusion. Cova et al. (2007) developed Swaddler which learns the relationships between application's execution points and internal states. Bisht et al. (2010b) proposed CANDID, which analyzes source code and retrofits them with additional candidate queries, against which runtime queries are matched. Wang and Li (2012a) developed SQLLEARN for learning normal SQL statements by program tracing techniques. Prevention based approaches generally require source-code access, and rebuilding the model upon changes to the application. Further, these are usually designed to protect only one web application, and suitable for a specific language and database platform.

2.4. Database firewall

Bertino et al. (2007) proposed profiling of database applications by encoding normal queries as a bit pattern and generating rule sets by association rule mining, deviations from which are detected as attacks. Kemalis and Tzouramanis (2008) developed SQL-IDS which generates specifications for normal queries defining their intended structure and employs validation at runtime to detect anomalies. These approaches require rebuilding the profile and specifications whenever the application is updated. Liu et al. (2009) designed SQLProb which dynamically extracts user inputs by pairwise sequence alignment and compares the parse trees, therefore it has high time complexity. Zhang et al. (2011) proposed TransSQL which maintains a duplicate of the database in Lightweight Directory Access Protocol (LDAP) form. Runtime

queries are automatically translated into LDAP requests and the results from SQL database and LDAP are compared to detect attacks. The approach is impracticable because every query is executed twice, and the LDAP database must be kept synchronized with the SQL database.

2.5. Database server

Low et al. (2002) developed DIDAFIT which fingerprints legitimate SQL queries with compact regular expressions during normal use, against which runtime queries are matched. The approach requires rebuilding the normal-use model whenever the web application is modified. Wei et al. (2006) proposed a technique for preventing SQLIA targeted specifically on stored procedures. The method combines static analysis of stored procedures and instrumenting them for runtime validation, therefore requires source code access. Kim and Lee (2014) proposed a data mining based approach using the internal query trees from the database server. This approach is applicable only to PostgreSQL database because other popular database servers do not provide access to the internally generated query trees.

2.6. Multi-layer approaches

Boyd and Keromytis (2004) developed SQLRand which appends a random key to every SQL keyword in the application code, and a proxy server between the application and database de-randomizes them into normal SQL queries. The solution requires modification of source code and parsing of randomized queries at the proxy. It is secured until the random key is unknown to attackers. Vigna et al. (2009) proposed combining an HTTP reverse proxy and an anomaly detector at the database level for reducing detection errors. Le et al. (2012) designed DoubleGuard based on a similar approach consisting of an IDS at the web server and another at the back-end database. Pinzón et al. (2013) proposed idMAS-SQL, a hierarchical multi-agent system monitoring at various layers. These approaches yield better accuracy at the cost of higher processing overhead, and are difficult to deploy, train and maintain.

3. The problem and motivation

An SQLIA attempt is successful only when the injected query gets executed on the database. The problem of SQL injection detection can therefore be stated as: "*Given an SQL query, determine if it is injected*." A query can be either genuine or injected, therefore detecting SQLIA is essentially a binary classification problem from the perspective of a database firewall. The problem becomes complex because the useful information available at this level is limited to: (1) the database on which the query is issued to execute, and (2) the SQL query itself.

Therefore, the only option is to analyze the incoming query by some technique and determine if it should be blocked from execution. Due to polymorphic nature of attack vectors, regular expressions or pattern matching do not suffice as practically feasible solutions.

Looking from a different angle, every SQL query is basically a string consisting of keywords, identifiers, operators, delimiters, literal values and other symbols. These constituent elements are referred to as *tokens*. Therefore, any SQL query, whether genuine or injected, is a sequence of tokens. Intuitively, the way these tokens are arranged, can provide valuable insight to identify malicious queries. The motivation behind this study is to view the tokens as *actors* and capture their *interaction* in form of a graph (or network). This is computationally simpler than generating the parse-tree of a query as per complex SQL grammar. Term graphs and co-occurrence networks are popular in text classification (Wang et al., 2005), keyword extraction (Palshikar, 2007), natural language processing (Mihalcea and Radev, 2011), information retrieval (Blanco and Lioma, 2012), and several other domains. Drawing inspiration from these, we intend to experimentally establish that, training SVM classifiers by the centrality of nodes is a simple yet powerful technique to identify SQL injection attacks at runtime with high degree of accuracy and minimal overhead on performance.

4. Proposed approach

The core of our approach consists of (1) converting an SQL query into a sequence of tokens preserving its structural composition, (2) generating a graph consisting of tokens as the nodes and interaction between them as weighted edges, (3) training an SVM classifier using the centrality measure of nodes, and (4) using the classifier to identify malicious queries at runtime. The rest of this section describes each of these components in detail.

4.1. Query tokenization

Kar et al. (2015) proposed a transformation scheme to convert SQL queries into a sentence like form facilitating their textual comparison in a vector space of terms. The scheme normalizes the identifiers, literal values, operators and all other symbols using capital alphabets A-Z separating the tokens with spaces. We incorporate minor modifications to the transformation scheme considering the following common techniques used by attackers to bypass detection:

1. Newline, carriage-return and tab characters in the query are replaced with normal space character (step 1 of Table 1) which neutralizes bypassing attempt by white-space spreading.

2. MySQL allows using reserved keywords as identifiers when delimited by the backquote (`) character. An attacker can unnecessarily delimit every identifier in the injection attack to bypass detection. As backquotes do not have any contribution towards the structural form of a query, they can be safely removed before substituting other symbols (see Table 2).

3. Parentheses are used to enclose function parameters and subqueries in SQL, but it is syntactically correct to use additional parenthesis-pairs even if not required. For example, `CHAR(65)` returns the character A, which can be augmented with extra parentheses as `CHAR((((65))))` producing the same return value. The expression `2 = 5 - 3` rewritten as `((2)) = (((5) - ((3))))` is still a tautology. This provides an opportunity to bypass detection by stuffing additional parenthesis pairs. However, attackers also sometimes inject one or two opening or closing parentheses in order to guess the parenthetical structure (if any) of the victim query. Therefore, matching parenthesis-pairs can be removed but any mismatching parentheses should be preserved and converted to tokens.

4. Attackers generally try to obfuscate by embedding empty comments within the injected code (e.g, `/**/OR/**/1/**/=/**/1`) to bypass detection. Also in MySQL, version specific commands can be written within inline comments. For example, in the query "`SELECT /*!50525 DISTINCT*/ retail_price FROM books`," the `DISTINCT` command will take effect on MySQL 5.5.25, but commented out on other versions. Therefore, empty comments (including those containing only white-spaces within) can be removed but non-empty comments must be preserved and tokenized.

For completeness, the modified transformation scheme is shown in Table 1. Substitutions for special characters and symbols is shown in Table 2. After normalization, we additionally perform the following post-processing steps to achieve uniformity among queries written using different referencing styles:

1. Substitute "`USRTBL DOT USRCOL`" by "`USRCOL`":- this normalizes queries written using *TableName.ColumnName* format to the general form.

2. Substitute "`CHR DOT USRCOL`" or "`STR DOT USRCOL`" by "`USRCOL`":- this normalizes queries or query segments using table aliases (e.g., *P.prodID* or *PR.product_id*)

3. Substitute "`ORDER BY STR`" by "`ORDER BY USRCOL`":- this normalizes queries where ordering of result is done over an aliased aggregated column.

Figure 2 shows examples of normalizing queries into a series of tokens. The original queries are intentionally shown in mixed case to show the effect of normalization. In the third query, backquotes and matching parentheses have been removed as per Table 2.

To visualize further, consider an injected query generated due to a bypass attempt (taken from the examples given in Section 1):

```
SELECT * FROM products WHERE prod_id = 24 OR 'DEF' > CoNcAt(ChAr(0x41),
cHaR(0x42), chAr(0x43));#
```

This query contains a number of symbols, operators and SQL function calls. The normalization scheme converts it into a sequence of tokens as:

```
SELECT STAR FROM USRTBL WHERE USRCOL EQ INT OR SQUT STR SQUT GT CONCAT
CHAR HEX CMMA CHAR HEX CMMA CHAR HEX SMCLN HASH
```

Any SQL query, irrespective of its length and complexity, is thus normalized into a sequence of tokens preserving its syntactical structure. A major benefit of query normalization is that, several queries which are different when compared as strings, get transformed into the same sequence of tokens. This generalization feature is very useful in substantially reducing the number of samples and processing overhead.

For MySQL version 5.5, the complete vocabulary including all keywords, functions, reserved words and the substitutions used in the transformation scheme consists of 686 distinct tokens. Each token is considered as an attribute (dimension) for presenting the dataset to SVM. We sort the tokens in alphabetic order for consistency of referencing.

4.2. Database enumeration

In order to normalize the identifiers used in the SQL query, such as database, table, and column names etc., all objects in the database server need to be enumerated. In a shared hosting environment, the database server may contain hundreds of databases, therefore on-demand enumeration is cumbersome and time consuming. To avoid this problem, we use a separate enumerator component which creates an XML schema of all database objects by querying the server's system catalog. A part of the XML schema is shown in Fig. 3.

For an incoming query, the name of the database on which it is issued for execution is known. The query tokenizer uses the corresponding section of the XML schema to normalize the identifiers. If any system objects have been referred in the query (commonly seen in injection attacks), the `<SystemDatabases>` section is used. As database schemata are altered by the developers as and when required, the XML schema is kept updated by running the enumerator at

regular intervals depending on how frequently the database objects are created or altered. On a shared server having large number of databases, the enumerator should be run more often, say every 30 minutes.

4.3. Graph of tokens

The normalization process converts a query into an ordered sequence of tokens $(t_1, t_2, \cdots, t_N)$. We generate a graph $G = (V, E, w)$ from the sequence which captures its structural properties as an interaction network of the tokens, facilitating graph analysis using quantitative metrics. The graph has *n* nodes, $n = |V|$ and $n \leq N$, corresponding to each unique token t_i, $i = 1 \ldots n$.

Definition 1. *A graph of tokens is a weighted graph* $G = (V, E, w)$ *where each vertex in* V *corresponds to a unique token in the normalized sequence,* $E \subset V^2$ *is the set of edges, and* $w : E \rightarrow \mathbb{N}$ *is a function that defines the weight of the edge. There is an edge between tokens* t_i *and* t_j *(usually* $i \neq j$*), with weight* w_{ij}*, if* t_i *and* t_j *occur within a sliding window spanning s tokens. If an edge between* t_i *and* t_j *already exists, its weight is incremented by the weight of the new edge.*

The weight of an edge indicates the strength of interaction between two tokens. Since the same token can occur more than once within the sliding window, self loops are allowed. Technically, such a graph is known as *multigraph* or *pseudograph*, but for simplicity we will refer to it as graph.

4.3.1. Undirected and directed graphs

The edges of the token graph can be either undirected or directed. In an undirected graph, the edges do not have any direction associated, while in a directed graph, the direction of the edge specifies a before-after relationship.

Definition 2. *In an undirected graph of tokens, there is an undirected edge between tokens* t_i *and* t_j*, with symmetric weight* $w_{ij} = w_{ji}$*, if* t_i *and* t_j *occur within a span of s tokens, irrespective of their order of occurrence.*

For the edges in a directed graph, we consider left-to-right order as it corresponds to the natural flow of tokens in SQL.

Definition 3. *In a directed graph of tokens, there is an edge $t_i \rightarrow t_j$, with weight w_{ij}, iff t_i occurs before t_j within a span of s tokens. The edges $t_i \rightarrow t_j$ and $t_j \rightarrow t_i$ are independently weighted, and normally $w_{ij} \neq w_{ji}$.*

In either case, as the sliding window proceeds, if an edge already exists, its weight is incremented by the value of the weight function. Fig. 4 illustrates the process of construction of undirected and directed graphs from a normalized query. It may be observed that, a self loop on USRCOL node occurs in the 5th move of the sliding window.

4.3.2. Uniform and proportional weighting

Co-occurrence of two tokens within the sliding window can be defined in terms of the *gap* between the tokens. The gap *g* between two tokens is the number of tokens present between them in the sequence. Two tokens are said to co-occur if $g \leq s - 2$, where *s* is the size of the sliding window. Therefore, $g = 0$ for consecutive tokens. Initially, there are only vertices but no edges in the graph, i.e., $w_{ij} = 0$, $\forall i, j$. Weighted edges are added as the window slides across the sequence. We define two types of weighting functions: uniform and proportional. Uniform weighting is given by:

$$w_{ij} = \begin{cases} 1 & \text{if} \quad 0 \leq g \leq s - 2 \\ 0 & \text{otherwise;} \end{cases}$$

Uniform weighting does not consider the relative distance between two tokens within the sliding window, i.e., occurrence of all tokens within the window are given equal importance. In proportional weighting, higher weight is assigned to the edge between tokens occurring closer to each other, given by:

$$w_{ij} = \begin{cases} s - g - 1 & \text{if} \quad 0 \leq g \leq s - 2 \\ 0 & \text{otherwise;} \end{cases}$$

In this case, the weight of the edge between the boundary tokens of the window is 1, and increases by 1 as the gap decreases, so that the weight of the edge between consecutive tokens is the highest. Fig. 5 illustrates uniform and proportional weighting methods. Proportional weighting captures more information than uniform weighting by considering the closeness between tokens and boosts the weight of edges between frequently co-occurring pairs.

4.3.3. Graph of tokens algorithm

The graph of tokens consisting of *n* nodes $(t_1, t_2, \cdots, t_n)$ is represented by an *adjacency* matrix **A** of size *n* × *n*. Rows and columns of the adjacency matrix are indexed by the names of the nodes (i.e., unique tokens), and each element $\mathbf{A}[t_i, t_j] = w_{ij}$. If there is no edge between t_i and t_j, then $w_{ij} = 0$. In an undirected graph, the adjacency matrix is diagonally symmetric. In case of directed graph, the rows are the source nodes and columns are target nodes. The adjacency matrix of a directed graph is generally asymmetric. Since we allow self loops, in both types of graphs, some of the diagonal elements $\mathbf{A}[t_i, t_i]$, $i = 1 \ldots n$, may be non-zero.

Undirected or directed graphs with uniform or proportional weighting, makes it possible to create four types of graphs: (1) undirected uniform weighted, (2) undirected proportional weighted, (3) directed uniform weighted, and (4) directed proportional weighted. A combined algorithm to generate these four types of graphs is presented in Algorithm 1. The algorithm takes the sequence of tokens *S*, size of sliding window *s*, and two additional parameters *G.type* and *W.mode* specifying the graph type and weighting method respectively.

Algorithm 1 Generate graph of tokens

Input: String of tokens S, window size s, $G.type$, $W.mode$
Output: Adjacency Matrix $\mathbf{A}$

```
1:  T[ ] ← SPLIT(S, space)
2:  N ← COUNT(T)
3:  V ← SORT(UNIQUE(T))
4:  n ← |V|
5:  for i = 1 to n do
6:      for j = 1 to n do
7:          A[t_i, t_j] ← 0
8:      end for
9:  end for
10: for i = 1 to N do
11:     if i + s ≤ N then
12:         p ← i + s
13:     else
14:         p ← N
15:     end if
16:     for j = i + 1 to p do
17:         if W.mode is Proportional then
18:             A[t_i, t_j] ← A[t_i, t_j] + i + s − j
19:         else
20:             A[t_i, t_j] ← A[t_i, t_j] + 1
21:         end if
22:         if G.type is Undirected then
23:             A[t_j, t_i] ← A[t_i, t_j]
24:         end if
25:     end for
26: end for
27: return A
```

The string of tokens is split into an array $T = (t_1, t_2, \cdots, t_N)$, which is made unique and sorted alphabetically (for consistency of referencing) to obtain the set of n vertices V. The elements of the adjacency matrix $\mathbf{A}[t_i,t_j]$, $i, j = 1 \ldots n$, are initialized to zero. The sliding window spanning s tokens moves from left to right by one token at a time until there are at least two tokens left, i.e., at least one edge is possible. Lines 11 to 15 prevent the token pointer p from moving beyond the end of the string. At every position of the sliding window, edges are considered between tokens occurring within it, and the weight as per the specified weighting method is added to the corresponding element of the adjacency matrix. For undirected graph, the matrix is made diagonally symmetric by copying $\mathbf{A}[t_i, t_j]$ to $\mathbf{A}[t_j, t_i]$. Finally, the algorithm returns the adjacency matrix.

4.3.4. Size of the sliding window

Size of the sliding window (s) influences the degree to which the interaction between the nodes is captured in the resulting graph. Tokens are considered as related to each other within the window, outside of which the relationship is not taken into account. A smaller window produces a sparse graph, while a larger window produces a dense one. Since our goal is to identify malicious queries using the graph properties, these must exhibit distinguishing features which an SVM classifier can be trained with. For information retrieval tasks in English language, Blanco and Lioma (2012) suggest the window in the range of 5 to 30, and recommend that 10 is a reasonable size for most purposes. From our initial experiments, we determined that for normalized SQL, window of size 3 to 7 perform comparatively better. We adopt the median value $s = 5$ for all experiments in this study.

4.3.5. Centrality measure

Generally, graph kernels are used in machine learning tasks which involve graphs as samples (Vishwanathan et al. 2010). However, graph kernels such as shortest paths or random walk, have the minimum time complexity of $O(n^3)$ and hence prohibitively expensive for a system intended to work at the database firewall layer. In graph theory and network analysis, node centrality is used to quantify relative importance of vertices in the graph. Centrality measures such as degree, betweenness, closeness etc., have been historically used in analysis of social networks (Freeman, 1979; Borgatti, 2005; Borgatti and Everett, 2006). Betweenness and closeness centralities are also expensive as they require graph traversal to find shortest paths. We therefore use degree centrality, the simplest and most fundamental of all, which can be directly computed from the adjacency matrix of the graph.

Degree centrality, or simply the degree, of a vertex v_i in a weighted graph $G = (V, E, w)$ is defined as the sum of weights of the edges incident upon it. Since self loops are allowed in graph of tokens, weight of the loop needs to be added twice while computing degree centrality. This is easily done by adding the diagonal element once to the row marginal. Denoting each element $\mathbf{A}[t_i, t_j]$ as a_{ij}, the degree centrality in an undirected graph is given by:

$$C_{D_i} = deg(v_i) = a_{ii} + \sum_{j=1}^{n} a_{ij} \qquad (1)$$

In case of directed graphs, indegree and outdegree of a vertex v_i are defined as the sum of weights of the edges directed to and from the vertex respectively. Weight of self loops are

considered once for both indegree and outdegree. The indegree and outdegree centralities, denoted by $C_{D_i}^{-}$ and $C_{D_i}^{+}$ respectively, are computed by the column and row marginals of the adjacency matrix **A** as:

$$C_{D_i}^{-} = deg^{-}(v_i) = \sum_{i=1}^{n} a_{ij} \quad (2)$$

$$\text{and} \quad C_{D_i}^{+} = deg^{+}(v_i) = \sum_{j=1}^{n} a_{ij} \quad (3)$$

The total degree of a node in a directed graph is the sum of indegree and outdegree, which is same as the degree centrality of the corresponding undirected graph. Therefore, we have:

$$C_{D_i} = deg(v_i) = deg^{-}(v_i) + deg^{+}(v_i) \quad (4)$$

4.4. Detection strategy

As indicated in Section 3, the only option for a detection system at the database firewall layer is to examine incoming SQL queries to identify injection attacks. In a shared hosting scenario where multiple web applications interface with a single database server, the system must be able to quickly analyze each query issued for execution. Since the injected code occurs only in part of a query, analyzing the entire query is a wastage of computational resource. Kar et al. (2015) showed that it is sufficient to examine only the `WHERE` clause portion of a query for detecting SQLIA. Liu et al. (2009) also compare the parse tree of the `WHERE` clauses in the input validation step. We adopt the same strategy of limiting the analysis to the `WHERE` clause to reduce the processing overhead. Hereafter, for ease of referencing, we shall use the term "*tail-end*" to mean the portion after the first `WHERE` keyword up to the end of an SQL query.

4.5. Graph of genuine vs. injected query

To demonstrate how the graphs capture interaction between the tokens, we provide undirected and directed graph examples of a genuine and an injected query. Consider the following queries:

```
Q1: SELECT * FROM products WHERE category_id = 5
    OR brand_id = 21 AND in_stock = 'Y'
Q2: SELECT * FROM vendors WHERE group_id = 10
    AND is_active = 'Y' OR 7 = 7
```

The query Q1 is a genuine query while Q2 is an injected query because it contains the tautological expression "`OR 7 = 7`". By normalizing the tail-ends (i.e., the portion after the `WHERE` keyword) of these queries, we get the following sequence of tokens respectively:

S1: USRCOL EQ INT OR USRCOL EQ INT AND USRCOL EQ SQUT CHR SQUT

S2: USRCOL EQ INT AND USRCOL EQ SQUT CHR SQUT OR INT EQ INT

Both the strings S1 and S2 contain the same unique tokens, with almost same frequency. Five tokens occur at the same position in both the strings. We construct the graph of tokens using proportional weighting as described in Section 4.3. The undirected and directed graphs are shown in Fig. 6 and Fig. 7 respectively. The thickness of the edges are approximately proportional to their weights for better visualization. Clearly, the graphs exhibit several distinguishable features, which can be expressed in terms of degree of the nodes for training an SVM classifier.

The frequency f_t and degree of tokens in undirected and directed graphs are shown in Table 3. The frequency of tokens in both strings are almost same, still the degrees are quite different. For directed graphs, in this example, it is interesting to observe that, the indegrees for the genuine query happen to be exactly same as the outdegrees for the injected query. Although this is only a coincidence, it illustrates how the interaction between tokens in a genuine query and an injected query are different by nature.

4.6. Support vector machine

Support Vector Machine (SVM) is a supervised machine learning technique introduced by Cortes and Vapnik (1995), which provides high accuracy dealing with high-dimensional data. SVM and its extensions have been extensively used for classification, regression and related problems in machine learning (Schölkopf and Smola, 2001).

In the most basic sense, SVM is a binary classifier that can assign samples into one of the two classes. Given l linearly separable training samples $\{\vec{x}_i, y_i\}$, $i = 1 \ldots l$, where each sample $\vec{x}_i$ has n attributes $(\vec{x}_i \in \mathbb{R}^n)$ and a class label $y_i \in \{+1, -1\}$, it finds the optimal hyperplane given by $\vec{w} \cdot \vec{x} + b = 0$ by maximizing the margin between the boundary vectors, so that $y_i(\vec{w} \cdot \vec{x}_i) \geq 1$ for $i = 1 \ldots l$, where $\vec{w}$ is a vector normal to the optimal hyperplane and b is the distance from origin. The linear classifier is given by $f(x) = \mathrm{sgn}(\vec{w} \cdot \vec{x} + b)$. The optimization problem of the soft-margin linear SVM is formulated as:

$$\begin{aligned} &\text{minimize} && \frac{1}{2}\|\vec{w}\|^2 + C\sum_{i=1}^{l}\xi_i \\ &\text{subjectto} && y_i(\vec{w}_i \cdot \vec{x}_i + b) \geq 1 - \xi_i, \quad \forall i \end{aligned} \tag{5}$$

where $\xi_i \geq 0$ are slack variables assigned to each sample and $C > 0$ is a trade-off parameter between the error and margin. If the data is not linearly separable in the input space, they are mapped into a higher dimensional feature space $\Phi(\vec{x}_i)$ where an optimal separating hyperplane can be found. The classifier is then given by:

$$f(\vec{x}) = \text{sgn}\left(\sum_{i=1}^{l}\alpha_i y_i K(\vec{x}_i, \vec{x}) + b\right) \tag{6}$$

where, α_i are the Lagrangian multipliers, and $K(\vec{x}_i, \vec{x}) = \Phi(\vec{x}_i) \cdot \Phi(\vec{x})$ is known as a kernel function. For linearly separable data, a kernel function can still be used by defining it as $K(\vec{x}_i, \vec{x}) = \vec{x}_i \cdot \vec{x}$. Other kernels, such as polynomial, radial basis function (RBF) and sigmoid etc., are used in classification problems employing SVM. We use the RBF kernel in our approach which is given by:

$$\begin{aligned} K(\vec{x}_i, \vec{x}) &= \exp\left(-\frac{\|\vec{x}_i - \vec{x}\|^2}{2\sigma^2}\right) \\ &= \exp\left(-\gamma\|\vec{x}_i - \vec{x}\|^2\right) \quad \text{where} \quad \gamma = \frac{1}{2\sigma^2} \end{aligned} \tag{7}$$

For the SVM solver in our prototype, we use LibSVM (Chang and Lin, 2011), which is an integrated software library for support vector classification, regression, and distribution estimation. A PHP wrapper built upon LibSVM is also available as a PECL Extension[4] making it easier to call LibSVM functions from within PHP code. LibSVM uses RBF kernel as the default kernel. During training of the SVM, appropriate values for C and γ are supplied.

4.7. The collision problem

Since we intend to use degree centrality of tokens to train an SVM, a natural question arises, "*What if the token graphs of a genuine and an injected query yield exactly the same degree for each node respectively?*" This would result in a collision of the vectors in the input space as well as the feature space. Although, theoretically it is possible that two different

[4]https://pecl.php.net/package/svm

adjacency matrices may produce the same corresponding row marginals, we show that the probability of such a collision is extraordinarily small.

Consider two token sequences S_a and S_b consisting of the same set of tokens $(t_1, t_2, \cdots, t_n)$, and $S_a \neq S_b$. Let the corresponding graphs G_a and G_b be represented by the adjacency matrices **A** and **B** of size $n \times n$. For simplicity, we consider graphs without self loops. Since $S_a \neq S_b$, we have $G_a \neq G_b$, and therefore **A** ≠ **B**. A collision of degrees between G_a and G_b arises if each row marginal of **A** is exactly same as that of the corresponding row of **B**, i.e.,

$$\sum_{j=1}^{n} a_{ij} = \sum_{j=1}^{n} b_{ij} \quad \forall\, i$$

Let k be the sum of elements of ith row of **A**, i.e., $a_{i1} + a_{i2} + \cdots + a_{in} = k$. Since $a_{ij} \in \mathbb{N}$, $k > 0$ and $a_{ij} < k, \forall\, j$. The number of ways n elements sum up to k is given by the combination ${}^{n+k-1}C_{n-1}$. In the matrix **B**, definitely $b_{ij} \leq k, \forall\, j$; otherwise the sum will be larger than k. The number of ways the ith row of **B** can be constructed is $(k+1)^n$. The probability that the ith row-sums of both **A** and **B** are equal to k, is then given by:

$$P_i \leq \frac{{}^{n+k-1}C_{n-1}}{(k+1)^n}$$

$$= \frac{(n+k-1)!}{(n-1)!\,k!\,(k+1)^n}$$

$$= \frac{(n+k-1)(n+k-2)\cdots(k+1)}{(n-1)!\,(k+1)^n}$$

Expanding the factorial and nth power in the denominator followed by a few rearrangement of terms, we get the following product sequence:

$$P_i \leq \frac{1}{(k+1)} \times \frac{k+1}{1(k+1)} \times \frac{k+2}{2(k+1)} \times \frac{k+3}{3(k+1)} \times \cdots$$

$$\ldots \times \frac{k+n-2}{(n-2)(k+1)} \times \frac{k+n-1}{(n-1)(k+1)}$$

For any $k > 0$, the first term is less than 1. The second term is equal to 1, therefore has no effect. From the third term onward, each fraction is less than 1, and becomes smaller and smaller as the denominator increases faster than the numerator. Therefore, the probability that the ith

rows of **A** and **B** sum up to the same value of k is very small. For example, if $n = 10$ and for ith row $k = 25$, the calculated probability comes to $\leq 3.716 \times 10^{-7}$.

The probability that all rows of **A** and **B** sum up to the same corresponding values is equal to product of the probabilities for each row, given by:

$$P = \prod_{i=1}^{n} P_i$$

Since each P_i is very small, the net probability is extremely small and beyond the required computational precision. Therefore, degree centrality of nodes from graph of tokens can be safely used to train an SVM without the risk of running into a collision situation within the scope of the problem.

5. System architecture

Being a supervised learning method, the SVM classifier needs to be trained first with a labeled dataset. Preparation of dataset has been discussed later in Section 6.1. SQLiGoT operates in two phases: (1) training phase, and (2) testing/runtime phase. The training phase begins with a labeled dataset containing genuine and injected queries. The tail-end of each query is extracted and normalized into sequence of tokens. The graph of tokens is generated and degree centrality of each token is computed, which are used to train the SVM classifier. During the testing/runtime phase, tail-end of the incoming query is extracted, normalized and graph is generated exactly in the same manner. The degree centralities are presented to the trained SVM classifier which predicts the class label. We use +1 for injected query (injection positive) and −1 for genuine query (injection negative). Based on the output from the classifier, the query is either rejected or forwarded to the database server for execution. Fig. 8 shows the generic architecture of the proposed system.

Using undirected or directed graph, and uniform or proportional weighting, the SVM classifier can be trained in several possible ways. The classifier can also consist of more than one SVM, each of which is trained in a different way, and the output of the individual SVMs are combined to derive the final output. Rest of this section describes different types of SVM classifier systems we created and tested in this study. The results have been discussed in Section 6.4.

5.1. Single SVM system

This is the conventional classifier design consisting of one SVM classifier. Using different combinations of graph type, weighting method and centrality measure, we tested the following six types of single SVM system:

1. Undirected graph, uniform weighting, degree centrality
2. Undirected graph, proportional weighting, degree centrality
3. Directed graph, uniform weighting, indegree centrality
4. Directed graph, proportional weighting, indegree centrality
5. Directed graph, uniform weighting, outdegree centrality
6. Directed graph, proportional weighting, outdegree centrality

In these single SVM systems, if the output of the classifier is +1, then the query is identified as an injected query and rejected. If the output is −1, then it is genuine and forwarded to the database server for execution.

5.2. Two SVM system

Using directed graphs, one SVM trained by indegree and another SVM trained by outdegree, are combined together to create a two-SVM classifier as shown in Fig. 9. The output of both SVMs (+1 or −1) are summed to get the final output. If the sum is +2, then both SVMs identify it as an injected query, so the query is rejected. If the sum is −2, then both SVMs agree that it is a genuine query, therefore it is forwarded to database server. The sum is zero when the predictions by the SVMs differ. In this case, the query is allowed to execute but logged as suspicious for examination by the DBA. The DBA examines the suspicious query log time to time and marks the queries as genuine or injected. These are then added to the training set and the SVMs are retrained, making the system more robust over time. We tested two types of such two-SVM systems using uniform and proportional weighting.

5.3. Three SVM system

As shown in Fig. 10, the classifier consists of three SVMs, which are trained with degree, indegree and outdegree from the directed graph of tokens. Degree centrality is normally computed from undirected graph, but it can be computed from the directed graph by Eq. (4). At runtime, the output of the three SVMs are summed, which can be either less than or greater than zero, but never equal to zero. If the sum is positive, it means that at least two of the three SVMs identify it as an injected query, so the query is rejected. If the sum is negative, then at least two SVMs predict it as a genuine query, so it is forwarded for execution. Thus, the final prediction is

done by a majority voting method. As the three-SVM system does not produce any indecisive output, involvement of DBA is not required. We tested three-SVM systems using both uniform and proportional weighting.

6. Experimental setup and evaluation

The experimental setup consisted of a standard desktop computer with Intel® Core-i3™ 2100 CPU @ 3.10 GHz and 2GB RAM, running CentOS 5.3 Server OS, Apache 2.2.3 web server with PHP 5.3.28 set up as Apache module, and MySQL 5.5.29 database server. Three client PCs of similar hardware configuration were connected through a passive switch to the server creating a 10/100 mbps LAN environment. Five web applications; namely, Bookstore (e-commerce), Forum, Classifieds, NewsPortal, and JobPortal were developed using PHP and MySQL with features and functionalities normally seen on real-world websites. The code of each web application was deliberately written *without* any input validation, ensuring that all web pages (including those requiring login) are entirely vulnerable to SQLIA. The web applications and their databases were set up on the server computer simulating a shared hosting scenario.

6.1. Dataset preparation

Training of the SVM classifier(s) requires an adequate collection of genuine and injected queries. Unfortunately, such a standard real-world dataset is not available. Researchers have used open source applications like phpBB, Wordpress, sample applications downloaded from GoToCode.com (offline since 3+ years), or synthetic queries. Some researchers have used the AMNESIA test bed[5] (Sun and Beznosov, 2008; Liu et al., 2009). The testbed consists of five sample web applications written in Java and requires Tomcat application server for deployment. We found that, each application consists of only a few webpages and database tables. Some authors have used only the `sqlmap`[6] tool on a sample application to generate the attack vectors (Wang and Li, 2012b; Kim and Lee, 2014; Kozik and Choraś, 2014), which may not adequately represent real-world attacks. For realistic simulation of attack scenarios, we decided to use as many as SQL injection tools available on the Internet along with manual attacks. We collected the dataset from the five web applications using a technique similar to honeypots. The General Query Log[7] option of MySQL (off by default) enables logging of all SQL queries received from

[5]http://www-bcf.usc.edu/ halfond/testbed.html
[6]http://sqlmap.org/
[7]http://dev.mysql.com/doc/refman/5.5/en/query-log.html

client applications. By switching it on, we extracted queries from the log files to prepare the dataset.

6.1.1. Collection of injected queries

SQL injection attacks were launched on the web applications using a number of automated vulnerability scanners and SQL injection tools downloaded from the Internet, such as, HP Scrawler, WebCruiser Pro, Wapiti, Skipfish, NetSparker (community edition), SQL Power Injector, BSQL Hacker, Simple SQLi Dumper, NTO SQL Invader, sqlmap, sqlsus, The Mole, darkMySQL, IronWasp, Havij, jSQL Injector, SQL Sentinel, and OWASP Zap. An advanced and versatile penetration testing platform based on Debian, known as *Kali Linux* (formerly BackTrack-5), is bundled with several additional scanners and database exploitation tools[8] such as grabber, bbqsql, nikto, w3af, vega, etc. Some SQL injection scripts were also downloaded from hacker and community sites such as Simple SQLi Dumper (Perl), darkMySQLi (Python), SQL Sentinel (Java), etc., and applied on the websites. Over a four month period, each tool was applied multiple times on the sample applications with different settings to ensure that widest possible varieties of injection patterns are captured. Pointers to each of these tools are not provided here due to brevity of space; the reader is advised to search on Google.

We also applied manual injection attacks following tips and tricks collected from various tutorials and black-hat websites. Since the web applications were intentionally made vulnerable, all injection attacks were successful. In the process, every SQL query issued by the web applications was logged by MySQL server in the general query log as planned. Over 9.03 million lines were written by MySQL into the log files, from which 6.75 million SQL queries were extracted. After removing duplicates, `INSERT` queries, and queries without `WHERE` clause, total 253,013 unique `SELECT`, `UPDATE`, and `DELETE` queries were obtained, containing a mixture of genuine as well as injected queries.

Each query was manually examined and the injected queries were carefully separated out. Total 59,811 injected queries were collected from the query set. It contained a good mix of all types of injection attacks, though the percentage of UNION-based, blind injection attacks, and time-based blind attacks was observed to be higher than others. In fact, these three SQL injection techniques are most commonly used by attackers. The tail-ends (i.e., the portion after the first `WHERE` keyword up to the end) of these queries were extracted producing 17,814 unique

[8]http://tools.kali.org/tools-listing

fragments. After normalization, 4,610 unique injection patterns (sequence of tokens) were obtained. The benefit of query normalization is evident as it reduced the size of samples by nearly 75%.

6.1.2. Collection of genuine queries

In the process of collecting injected queries, many genuine queries were also collected side by side, and more were collected by normal browsing the web applications. We used Xenu Link Sleuth[9] to quickly simulate browsing of the web applications. However, contrary to our expectation, we obtained only 773 unique genuine sequences from the logged queries. This is because, most of the queries in the sample web applications had simple `WHERE` clauses. Since our approach focuses on the structural composition of the tail-end of a query irrespective of the source application, it really does not matter from which web applications we collect the samples. We contacted two local web development companies working in PHP/MySQL and obtained nearly 13.5GB of MySQL log files from their staging servers, containing 6.15 million queries. Total 3,457 new genuine patterns were obtained from the normalized tail-ends. A few injected queries were also spotted in the logs, but the patterns were already there in the injected set. Another 654 genuine patterns were synthetically generated according to SQL grammar, totaling it up to 4,884.

6.2. Scaling of degree centralities

The 686 tokens in the vocabulary constitute the attributes for each sample vector. We found that only 189 tokens are in use in our dataset. However, tokens present in a runtime query cannot be predicted apriori, therefore we do not apply any feature selection. The tokens are sorted alphabetically and the position in the list is considered as the attribute ID. For example, the degree centralities of tokens in S1 (see Table 3) are (AND:20 CHR:14 EQ:53 INT:37 OR:18 SQUT:27 USRCOL:50). Using the attribute IDs from the sorted list of tokens, the data vector is (16:20 70:14 186:53 276:37 428:18 561:27 646:50). LibSVM recommends that the attribute values in the training data are scaled in the range [0,1] (Hsu et al., 2003). There are two ways this can be achieved: (1) by converting the degree vector into an unit vector, or (2) by expressing the degree as a relative degree centrality. We prefer the later as it is computationally cheaper. Relative degree centrality is given by:

[9]http://home.snafu.de/tilman/xenulink.html -- A broken link checker

$$C'_{D_i} = \frac{C_{D_i}}{max(C_D)} \qquad (8)$$

Hence for any graph, the token having the highest absolute degree, always has a relative degree centrality of 1.0. The degrees of other tokens are accordingly scaled in the range [0, 1]. In the above example, dividing all degrees by the highest degree, we get the scaled data as (16:0.3774 70:0.2642 186:1.0000 276:0.6981 428:0.3585 561:0.5094 646:0.9434) as required by LibSVM.

6.3. Training of SVM classifier

Our final dataset consisted of 4610 injected sequences and 4884 genuine sequences, therefore well balanced. Due to query normalization, the dataset actually represents large number of queries. For each of the SVM classifiers, the data in LibSVM format was generated using the type of graph and weighting method as applicable. Class label +1 was used for injected queries and −1 for genuine queries. We used the default C-SVC classifier with RBF kernel for training the SVMs. The parameters C and γ were determined by the *grid search* tool provided with LibSVM using default 5-fold cross-validation. For training data using degree and indegree, best value of C and γ were determined as 32.0 and 0.5 respectively. For outdegree, the best value of C came out as 128.0, but value of γ remained unchanged at 0.5. Fig. 11 shows the gnuplot output of the grid search tool for a single SVM classifier using degree centrality from undirected graph with proportional weighting.

Fig. 12 shows the learning curve of the above single SVM classifier with respect to percentage of samples used for training. It is observed that, with only 5% of the data randomly selected for training and rest 95% used for testing, the system gives 96.23% accuracy with 4.17% false positive rate. This is very encouraging and affirms efficacy of the proposed technique. As the percentage of training samples is increased, the accuracy increases at a higher rate until 60% and then slows down. Accordingly, the false positive rate decreases at a higher rate and then slows down after 70%.

6.4. Experimental results

As the SVM parameters (C and γ) are determined using 5-fold cross-validation, we choose the training and testing sets in the same ratio for all experiments, i.e., 80% for training and 20% for testing. Thus the training set contains 7595 (3907 genuine + 3688 injected) sequences and the testing set contains 1899 (977 genuine + 922 injected) sequences. All systems

were tested 100 times with randomly selected training and testing sets, and average results were computed along with the standard deviations. The training time of the SVMs was found to vary from 670.80 ms to 873.60 ms.

Results of the six types of single SVM systems (refer Section 5.1) are presented in Table 4. For undirected graphs using degree centrality, outcome of proportional weighing method is better than uniform weighting. Among the four systems with directed graphs, proportional weighting and outdegree centrality provides the best result. In all systems, proportional weighting performs better than uniform weighting, because proportional weighting also captures information about relative distance between tokens within the sliding window. We also find that, for both weighting methods, undirected graphs using degree centrality perform better than directed graphs using indegree or outdegree centrality. This is because of loss of information when either indegree or outdegree is used to train the SVM. In case of directed graphs, outdegree is found to perform better than indegree in both weighting methods. The natural flow of SQL language from left-to-right could be a possible explanation for this. Though all systems produce very good level of accuracy and low false positive rate, undirected graph with proportional weighting and degree centrality turns out to be the winner with 99.47% accuracy and 0.31% false positive rate.

Table 5 presents the results of two types of two-SVM systems discussed in Section 5.2, using uniform and proportional weighting. Recall that when the predictions by the two SVMs differ, the query is logged as suspicious to be examined by the DBA. The columns SUS and SPR refer to number of queries logged as suspicious and the suspicion rate respectively. We find that, proportional weighting performs marginally better than uniform weighting. The suspicion rate for both is 1.26%. Examining the suspicious log, we found that it mostly contained syntactically incorrect or illegal queries.

We also tested two types of three-SVM systems described in Section 5.3 using both weighting methods. The results are presented in Table 6. In this case also, proportional weighting performed better than uniform weighting, and the accuracy surpasses the best single-SVM system.

Comparison of the best results of the proposed SVM systems is shown in Table 7. Though the accuracy of two-SVM system is highest at 99.73%, it produces 1.26% suspicious queries and requires DBA involvement. The three-SVM system produces the best accuracy of

99.63%, but it requires more computational time and memory space. The computational time can be reduced by running the three SVMs in parallel. The false positive rates of all systems are same at 0.31%. Considering the accuracy and simplicity of single SVM system, it is most suitable for a shared hosting environment.

To ascertain effectiveness of the approach in a practical scenario, we tested it on the five sample web applications by launching SQL injection attacks using the automated tools mentioned in Section 6.1.1. The three-SVM system using proportional weighting (Section 5.3) was used for the tests. Queries predicted as injected by SQLiGoT were also allowed to execute, so that the automated tools can smoothly proceed on performing the series of attacks they are programmed for. The SQL queries received by SQLiGoT, normalized tail-ends, and output of the classifier were logged. Table 8 shows the results compiled from the log file. The results confirm that the system practically performs well with very good accuracy and low false positive rate.

We believe there is still scope to improve the accuracy by fine tuning of the SVM training parameters. Since our focus is to establish that, using degree centrality from graph of tokens is a practically feasible technique to identify injection attacks, we rely on parameters suggested by coarse grid search and use the defaults provided by LibSVM. For SQLIA detection at the database firewall level, particularly in a shared hosting scenario, web application independence and speed is more important as long as misclassifications are within acceptable limits. As the dataset was prepared from very large number of legitimate queries and injection attacks using several automated attack tools, the experimental results demonstrate that the objectives of the study are well realized.

6.5. Performance overhead

The processing overhead added by SQLiGoT at runtime consists of four components: (1) extracting the tail-end of the incoming query, (2) normalizing into sequence of tokens, (3) generating graph of tokens & computing degree centrality, and (4) predicting the class label by the SVM classifier. Out of these, extracting the tail-end takes negligible amount of time; so it is ignored while calculating the net impact.

Time consumed for normalizing a query into sequence of tokens was measured by applying it on large number of randomly selected queries. The average time required was found to be 0.983ms per query. Generating the graph of tokens and computing degree centrality was also measured in a similar manner and found out to be 0.644 ms. Time for class prediction by

one SVM was observed to vary from 0.352 ms to 0.984 ms, averaging to 0.668 ms. Thus the average processing time per incoming query comes to 2.295 ms. Even if a web page of a real-world dynamic website issues as many as 20 SQL queries for execution, the total theoretical delay introduced by SQLiGoT is within 50 ms over each page load. Normally, the page load times on the Internet are in order of multiple seconds, hence a delay of 50 ms is hardly perceivable by the end user.

Impact on performance under practical usage scenario was assessed by conducting stress testing of the web applications. We used Pylot[10] and Siege[11] which are free load testing and benchmarking tools. The tests were conducted with and without SQLiGoT (single SVM) activated for 30 minutes on each web application with 10 to 100 concurrent users, 10 ms request interval, and 10 seconds ramp-up time. The average difference in response times for each application is shown in Fig. 13. The delay is proportional to number of queries issued by the web pages and increases with number of concurrent users. Overall, the performance impact is found to be nearly 2.5% the average response time of the web server.

6.6. Comparison with existing methods

Though many detection or prevention techniques have been acknowledged in the literature, only some of them have been implemented from perspective of practicality. Most of the approaches have been validated using synthetic dataset or sample applications mimicking near real-world scenarios. Since each approach has its advantages and disadvantages under specific scenarios, it is difficult to do a comparison based on empirical results. Therefore, we analytically compare our approach with other techniques we studied. The comparison is shown in Table 9, which is based on the following aspects:

1. *Specific coding method*: the approach requires a specific coding method or use of a programming framework.
2. *Source code access*: the approach requires access to the source code or instruments it with additional code.
3. *Platform specific*: the approach is applicable or suitable only for a specific programming language and/or database platform.

[10] http://www.pylot.org/
[11] http://www.joedog.org/siege-home/

4. *Normal-use modeling*: the approach requires building a model of the SQL queries generated by the web application in a secured environment.

5. *Multiple websites*: the approach can protect multiple web applications hosted on a shared server.

6. *Time complexity*: the general time complexity of the system – high or low.

7. *Practical usability*: how well the approach is practically usable in a real production environment.

We also compare our approach with these approaches based on the ability to prevent or detect various types of SQL injection attacks as shown in Table 10. Though SQLiGoT detected all of the tautological attack vectors in our dataset and subsequent tests, we show it as partial because theoretically tautological expressions can be formed in infinite number of ways.

7. Related work

To the best of our knowledge, graph of tokens have not been proposed for detection of SQLIA in the literature so far. A few studies have proposed SVM-based systems for identifying malicious queries including SQLIA, which can be considered as related to our work to some extent.

Bockermann et al. (2009) proposed SVM learning of normal SQL queries using context-sensitive tree-kernels to detect anomalous queries at runtime. The approach has high time complexity due to parse tree generation. Choi et al. (2011) attempted to detect malicious code injection by training an SVM using N-Grams extracted from queries. The approach ignores symbols and operators while extracting N-Grams, which are important syntactic elements in a query. Wang and Li (2012a) proposed to train an SVM classifier by attaching the program trace with every SQL query. They used a combination of tree kernel and string similarity by Levenshtein distance. The approach requires access to source code, and upon modification of the application, the models must be regenerated. Kim and Lee (2014) used internal query trees from database log to train an SVM classifier. They first generate an intermediate representation of the query tree as a multi-dimensional sequence and then convert it into a feature vector, which takes about 503 ms per query. The approach is applicable only to PostgreSQL database because other database servers do not log or provide access to the internal query trees.

8. Conclusions and future work

This paper presented a novel approach to detect SQL injection attacks by modeling SQL queries as graph of tokens and using centrality of nodes to train an SVM classifier. We normalized SQL queries into sequence of tokens preserving the structural composition and captured the interaction between the tokens in form of a graph. The approach was designed to work at the database firewall layer and was implemented in a prototype named SQLiGoT. The system was exhaustively tested using undirected and directed graphs with two different edge-weighting methods. Alternative designs of the SVM classifier consisting of single and multiple SVMs were also proposed, tested, and compared. The experimental results obtained on five fully vulnerable web applications confirm the effectiveness of our approach. The system does not require building a normal-use model of queries, nor requires access to the source code. It can protect multiple web applications hosted on a shared server, giving it an edge over existing methods. The performance overhead is also imperceptible for the end-users. The approach uses features available in most modern programming languages and can be ported to other platforms without requiring major modifications.

As speed is already taken care of, in future we will focus on improving the accuracy by employing a feature selection method and fine tuning of SVM classifiers. We will also examine feasibility of other kernels described in the literature. Next we plan to implement automatic re-training of the system when new genuine or attack sequences outside the original dataset are detected. Distributed installations of SQLiGoT, communicating with each other to keep their models up to date, could also be another interesting possibility to work on in the future.

References

Alserhani F, Akhlaq M, Awan I, Cullen A. Event-based Alert Correlation System to Detect SQLI Activities. In: Advanced Information Networking and Applications (AINA), 2011 IEEE International Conference on. IEEE; 2011. p. 175–82.

Benedikt M, Freire J, Godefroid P. VeriWeb: Automatically Testing Dynamic Web Sites. In: In Proceedings of 11th International World Wide Web Conference (WWW'2002). Citeseer; 2002.

Bertino E, Kamra A, Early J. Profiling Database Application to Detect SQL Injection Attacks. In: Performance, Computing, and Communications Conference, 2007. IPCCC 2007. IEEE Internationa. IEEE; 2007. p. 449–58.

Bisht P, Hinrichs T, Skrupsky N, Bobrowicz R, Venkatakrishnan V. NoTamper: Automatic Blackbox Detection of Parameter Tampering Opportunities in Web Applications. In: Proceedings of the 17th ACM conference on Computer and communications security. ACM; 2010a. p. 607–18.

Bisht P, Madhusudan P, Venkatakrishnan V. CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks. ACM Transactions on Information and System Security (TISSEC) 2010b;13(2):14.

Bisht P, Sistla A, Venkatakrishnan V. Automatically Preparing Safe SQL Queries. Financial Cryptography and Data Security 2010c;:272–88.

Blanco R, Lioma C. Graph-based term weighting for information retrieval. Information retrieval 2012;15(1):54–92.

Bockermann C, Apel M, Meier M. Learning sql for database intrusion detection using context-sensitive modelling. In: Detection of Intrusions and Malware, and Vulnerability Assessment. Springer; 2009. p. 196–205.

Borgatti SP. Centrality and network ow. Social networks 2005;27(1):55–71.

Borgatti SP, Everett MG. A graph-theoretic perspective on centrality. Social networks 2006;28(4):466–84.

Boyd S, Keromytis A. SQLrand: Preventing SQL injection attacks. In: Applied Cryptography and Network Security. Springer; 2004. p. 292–302.

Buehrer G, Weide B, Sivilotti P. Using Parse Tree Validation to Prevent SQL Injection Attacks. In: Proceedings of the 5th International Workshop on Software Engineering and Middleware. ACM; 2005. p. 106–13.

Carettoni L, Paola SD. HTTP Parameter Pollution. `https://www.owasp.org/images/b/ba/AppsecEU09_CarettoniDiPaola_v0.8.pdf`; 2009. Accessed: 2013-09-18.

Chang CC, Lin CJ. Libsvm: A library for support vector machines. ACM Transactions on Intelligent Systems and Technology (TIST) 2011;2(3):27.

Choi J, Kim H, Choi C, Kim P. Efficient Malicious Code Detection Using N-Gram Analysis and SVM. In: Network-Based Information Systems (NBiS), 2011 14th International Conference on. IEEE; 2011. p. 618–21.

Cortes C, Vapnik V. Support-vector networks. Machine learning 1995;20(3):273–97.

Cova M, Balzarotti D, Felmetsger V, Vigna G. Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications. In: Recent Advances in Intrusion Detection. Springer; 2007. p. 63–86.

Dahse J. Exploiting hard filtered SQL Injections. http://websec.wordpress.com/2010/03/19/exploiting-hard-filtered-sql-injections/; 2010. Accessed: 2011-06-23.

Freeman LC. Centrality in social networks conceptual clarification. Social networks 1979;1(3):215–39.

Gauci S, Henrique WG. WAFW00F identifies and fingerprints Web Application Firewall (WAF) products. https://github.com/sandrogauci/wafw00f; 2014. Accessed: 2015-03-27.

Halfond W, Orso A. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. ACM; 2005. p. 174–83.

Halfond W, Viegas J, Orso A. A Classification of SQL-injection Attacks and Countermeasures. In: International Symposium on Secure Software Engineering (ISSSE). 2006. p. 12–23.

Hsu CW, Chang CC, Lin CJ, et al. A Practical Guide to Support Vector Classication. http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf; 2003. Accessed: 2015-03-18.

Jovanovic N, Kruegel C, Kirda E. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In: Security and Privacy, 2006 IEEE Symposium on. IEEE; 2006. p. 257–63.

Kar D, Panigrahi S, Sundararajan S. SQLiDDS: SQL Injection Detection Using Query Transformation and Document Similarity. In: Distributed Computing and Internet Technology. Springer; 2015. p. 377–90.

Kemalis K, Tzouramanis T. SQL-IDS: A Specification-based Approach for SQL-injection Detection. In: Proceedings of the 2008 ACM symposium on Applied computing. ACM; 2008. p. 2153–8.

Kim MY, Lee DH. Data-mining based SQL Injection Attack Detection using Internal Query Trees. Expert Systems with Applications 2014;41(11):5416–30.

Kozik R, Choraś M. Machine Learning Techniques for Cyber Attacks Detection. In: Image Processing and Communications Challenges 5. Springer; 2014. p. 391–8.

Le M, Stavrou A, Kang BB. Doubleguard: Detecting intrusions in multitier web applications. Dependable and Secure Computing, IEEE Transactions on 2012;9(4):512–25.

Liu A, Yuan Y, Wijesekera D, Stavrou A. SQLProb: A Proxy-based Architecture towards Preventing SQL Injection Attacks. In: Proceedings of the 2009 ACM symposium on Applied Computing. ACM; 2009. p. 2054–61.

Long J, Gardner B, Brown J. Google hacking for penetration testers. volume 2. Syngress, 2011.

Low W, Lee J, Teoh P. DIDAFIT: Detecting Intrusions in Databases through Fingerprinting Transactions. In: Proceedings of the 4th International Conference on Enterprise Information Systems (ICEIS). Citeseer; volume 264; 2002. p. 265–7.

Lupták P. Bypassing Web Application Firewalls. In: Proceedings of 6th International Scientific Conference on Security and Protection of Information. University of Defence, Czech Republic; 2011. p. 79–88. Accessed: 2014-02-13.

Maciejak D, Lovet G. Botnet-Powered Sql Injection Attacks: A Deeper Look Within. In: Virus Bulletin Conference. 2009. p. 286–8.

Makiou A, Begriche Y, Serhrouchni A. Improving web application firewalls to detect advanced sql injection attacks. In: Information Assurance and Security (IAS), 2014 10th International Conference on. IEEE; 2014. p. 35–40.

Maor O, Shulman A. SQL Injection Signatures Evasion (White paper). Imperva Inc 2004;http://www.issa-sac.org/info_resources/ISSA_20050519_iMperva_SQLInjection.pdf.

McClure R, Kruger I. SQL DOM: Compile Time Checking of Dynamic SQL Statements. In: Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on. IEEE; 2005. p. 88–96.

Mihalcea R, Radev D. Graph-based natural language processing and information retrieval. Cambridge University Press, 2011.

Moosa A. Artificial neural network based web application firewall for sql injection. World Academy of Science, Engineering & Technology 2010;64:12–21.

Palshikar GK. Keyword extraction from a single document using centrality measures. In: Pattern Recognition and Machine Intelligence. Springer; 2007. p. 503–10.

Pinzón CI, De Paz JF, Herrero A, Corchado E, Bajo J, Corchado JM. idmas-sql: intrusion detection based on mas to detect and block sql injection through data mining. Information Sciences 2013;231:15–31.

Quartini S, Rondini M. Blind Sql Injection with Regular Expressions Attack. https://www.exploit-db.com/docs/17397.pdf; 2011. Accessed: 2013-02-11.

Ruse M, Sarkar T, Basu S. Analysis & Detection of SQL Injection Vulnerabilities via Automatic Test Case Generation of Programs. In: Applications and the Internet (SAINT), 2010 10th IEEE/IPSJ International Symposium on. IEEE; 2010. p. 31–7.

Schmitt I, Schinzel S. WAFFle: Fingerprinting Filter Rules of Web Application Firewalls. In: Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT'12). 2012. p. 34–40.

Schölkopf B, Smola AJ. Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond. Cambridge, MA, USA: MIT Press, 2001.

Scholte T, Balzarotti D, Kirda E. Have Things Changed Now? An Empirical Study on Input Validation Vulnerabilities in Web Applications. Computers & Security 2012a.

Scholte T, Robertson W, Balzarotti D, Kirda E. Preventing input validation vulnerabilities in web applications through automated type analysis. In: Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual. IEEE; 2012b. p. 233–43.

Shahriar H, North S, Chen WC. Client-side detection of sql injection attack. In: Advanced Information Systems Engineering Workshops. Springer; 2013. p. 512–7.

Shin Y, Williams L, Xie T. SQLUnitgen: Test Case Generation for SQL Injection Detection. North Carolina State University, Raleigh Technical Report, NCSU CSC TR 2006;21:2006.

Sun ST, Beznosov K. SQLPrevent: Effective Dynamic Detection and Prevention of SQL Injection Attacks Without Access to the Application Source Code. Technical Report; Tech. Rep. LERSSE-TR-2008-01, Laboratory for Education and Research in Secure Systems Engineering, University of British Columbia; 2008.

Thomas S, Williams L, Xie T. On Automated Prepared Statement Generation to Remove SQL Injection Vulnerabilities. Information and Software Technology 2009;51(3):589–98.

TrustWave. Executive Summary: Trustwave 2013 Global Security Report. https://www.trustwave.com/global-security-report; 2013. Accessed: 2014-08-17.

TrustWave. Trustwave 2015 Global Security Report. https://www2.trustwave.com/rs/815-RFM-693/images/2015_TrustwaveGlobalSecurityReport.pdf; 2015. Accessed: 2015-04-10.

Vigna G, Valeur F, Balzarotti D, Robertson W, Kruegel C, Kirda E. Reducing Errors in the Anomaly-based Detection of Web-based Attacks through the Combined Analysis of Web Requests and SQL Queries. Journal of Computer Security 2009;17(3):305–29.

Vishwanathan SVN, Schraudolph NN, Kondor R, Borgwardt KM. Graph kernels. The Journal of Machine Learning Research 2010;11:1201–42.

Wang W, Do DB, Lin X. Term graph model for text classification. In: Advanced Data Mining and Applications. Springer; 2005. p. 19–30.

Wang Y, Li Z. SQL Injection Detection via Program Tracing and Machine Learning. In: Internet and Distributed Computing Systems. Springer; 2012a. p. 264–74.

Wang Y, Li Z. SQL Injection Detection via Program Tracing and Machine Learning. In: Internet and Distributed Computing Systems. Springer; 2012b. p. 264–74.

Wassermann G, Yu D, Chander A, Dhurjati D, Inamura H, Su Z. Dynamic Test Input Generation for Web Applications. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis. ACM; 2008. p. 249–60.

Wei K, Muthuprasanna M, Kothari S. Preventing SQL Injection Attacks in Stored Procedures. In: Software Engineering Conference, 2006. Australian. IEEE; 2006. p. 191–8.

Zhang K, Lin C, Chen S, Hwang Y, Huang H, Hsu F. TransSQL: A Translation and Validation-Based Solution for SQL-injection Attacks. In: Robot, Vision and Signal Processing (RVSP), 2011 First International Conference on. IEEE; 2011. p. 248–51.

Figure 1: Multi-tier architecture of web applications

Figure 2: Examples of SQL query tokenization

Figure 3: Partial XML schema by database enumerator

Figure 4: Construction of *graph-of-tokens*: (a) undirected, (b) directed

Figure 5: Weighting of edges: (a) uniform, (b) proportional

Figure 6: Undirected graph of tokens

Figure 7: Directed graph of tokens

Figure 8: Generic architecture of SQLiGoT system

Figure 9: Two SVM system

Figure 10: Three SVM system

Figure 11: Finding best *C* and γ by grid search

Figure 12: Learning curve of a single SVM classifier

Figure 13: Performance overhead of SQLiGoT (single SVM)

Table 1: The query normalization scheme

Step	Token/Symbol	Substitution
1.	White-space characters (\r, \n, \t)	Space
2.	Anything within single/double quotes	
	(a) Hexadecimal value	HEX
	(b) Decimal value	DEC
	(c) Integer value	INT
	(d) IP address	IPADDR
	(e) Single alphabet character	CHR
	(f) General string (none of the above)	STR
3.	Anything outside single/double quotes	
	(a) Hexadecimal value	HEX
	(b) Decimal value	DEC
	(c) Integer value	INT
	(d) IP address	IPADDR
4.	System objects	
	(a) System databases	SYSDB
	(b) System tables	SYSTBL
	(c) System table column	SYSCOL
	(d) System variable	SYSVAR
	(e) System views	SYSVW
	(f) System stored procedure	SYSPROC
5.	User-defined objects	
	(a) User databases	USRDB

	(b) User tables	USRTBL
	(c) User table column	USRCOL
	(d) User-defined views	USRVW
	(e) User-defined stored procedures	USRPROC
	(f) User-defined functions	USRFUNC
6.	SQL keywords, functions and reserved words	To uppercase
7.	Any token not substituted so far	
	(a) Single alphabet	CHR
	(b) Alpha-numeric without space	STR
8.	Other symbols and special characters	As per Table 2
9.	The entire query	To uppercase
10.	Multiple spaces	Single space

Table 2: Tokenization of special characters (Step–8 of Table 1)

Symbol	Name	Substitution
`	Backquote	Remove
/**/	Empty comment	Remove
! = or <>	Not Equals	NEQ
&&	Logical AND	AND
||	Logical OR	OR
/*	Comment start	CMTST
*/	Comment end	CMTEND
~	Tilde	TLDE
!	Exclamation	EXCLM
@	At-the-rate	ATR
#	Pound	HASH
$	Dollar	DLLR
%	Percent	PRCNT
^	Caret	XOR
&	Ampersand	BITAND

|	Pipe or bar	BITOR
*	Asterisk	STAR
-	Hyphen/minus	MINUS
+	Addition/plus	PLUS
=	Equals	EQ
( )	Matching parentheses	Remove
(	Orphan opening parenthesis	LPRN
)	Orphan closing parenthesis	RPRN
{	Opening brace	LCBR
}	Closing brace	RCBR
[	Opening bracket	LSQBR
]	Closing bracket	RSQBR
\	Back slash	BSLSH
:	Colon	CLN
;	Semi-colon	SMCLN
"	Double quote	DQUT
'	Single quote	SQUT
<	Less than	LT
>	Greater than	GT
,	Comma	CMMA
.	Stop or period	DOT
?	Question mark	QSTN
/	Forward slash	SLSH

Table 3: Degree of nodes in graph of tokens

	Undirected graph				Directed graph			
	Genuine (S1)		Injected (S2)		Genuine (S1)		Injected (S2)	
Token	f_t	C_D	f_t	C_D	C_D^-	C_D^+	C_D^-	C_D^+
AND	1	20	1	19	10	10	9	10
CHR	1	14	1	20	10	4	10	10

EQ	3	53	3	48	24	29	24	24
INT	2	37	3	44	17	20	27	17
OR	1	19	1	19	9	10	10	9
SQUT	2	27	2	40	20	7	20	20
USRCOL	3	50	2	30	20	30	10	20

Table 4: Experimental results of single SVM systems

Graph Type	Weighting	Centrality	TP	FN	TN	FP	Precision	Recall	FPR	Accuracy	F1-Score
Undirected	Uniform	Degree	914	8	973	4	99.56 %	99.13 %	0.41 %	99.37 %	99.35 %
		Std. Dev. (σ)	3.10		2.77		0.27	0.38	0.26	0.17	0.18
Undirected	Proportional	Degree	916	7	973	3	99.67 %	99.24 %	0.31 %	99.47 %	99.46 %
		Std. Dev. (σ)	2.33		2.09		0.21	0.23	0.22	0.15	0.14
Directed	Uniform	Indegree	914	8	965	12	98.70 %	99.13 %	1.23 %	98.95 %	98.92 %
		Std. Dev. (σ)	3.15		4.04		0.43	0.34	0.41	0.25	0.25
Directed	Proportional	Indegree	916	6	966	11	98.81 %	99.35 %	1.13 %	99.10 %	99.08 %
		Std. Dev. (σ)	2.63		3.04		0.32	0.29	0.31	0.19	0.20
Directed	Uniform	Outdegr	91	8	96	9	99.02	99.13	0.92	99.10	99.08

		ee	4		8		%	%	%	%	%
		Std. Dev. (σ)	3.42		3.33		0.31	0.37	0.29	0.26	0.26
Directed	Proportional	Outdegree	915	7	969	8	99.13 %	99.24 %	0.82 %	99.21 %	99.19 %
		Std. Dev. (σ)	3.16		2.83		0.36	0.34	0.34	0.22	0.23

Table 5: Results of Two-SVM systems using directed graph

Weighting	TP	FN	TN	FP	SUS	SPR	Precision	Recall	FPR	Accuracy	F1-Score
Uniform	908	2	961	4	24	1.26 %	99.56%	99.78 %	0.41 %	99.68%	99.67%
Std. Dev. (σ)	4.47	1.43	4.15	1.91	5.10	0.27	0.21	0.15	0.20	0.13	0.13
Proportional	908	2	962	3	24	1.26 %	99.67%	99.78 %	0.31 %	99.73%	99.73%
Std. Dev. (σ)	3.89	1.35	3.93	1.76	4.73	0.25	0.19	0.16	0.18	0.12	0.12

Table 6: Results of three-SVM systems using directed graph

Weighting	TP	FN	TN	FP	Precision	Recall	FPR	Accuracy	F1-Score
Uniform	916	6	971	6	99.35%	99.35%	0.61%	99.37%	99.35%
Std. Dev. (σ)	2.74		2.44		0.26	0.30	0.25	0.18	0.19
Proportional	918	4	974	3	99.67%	99.57%	0.31%	99.63%	99.62%
Std. Dev. (σ)	2.46		2.38		0.24	0.27	0.24	0.17	0.18

Table 7: Best results of different types of SVM systems (proportional weighting)

SVM System	Graph	Centrality	Precision	Recall	FPR	SPR	Accuracy	F1-Score
Single SVM	Undirected	Degree	99.67%	99.24%	0.31%	–	99.47%	99.46%
Two SVM	Directed	Indegree, Outdegree	99.67%	99.78%	0.31%	1.26%	99.73%	99.73%
Three SVM	Directed	Degree, Indegree, Outdegree	99.67%	99.57%	0.31%	–	99.63%	99.62%

Table 8: Test results on sample applications using three-SVM system

Application	Queries	TP	FN	FP	TN	Precision	Recall	FPR	Accuracy	F1Score
Bookstore	4468	3876	9	2	581	99.95%	99.77%	0.34%	99.75%	99.86%
Forum	3617	3046	7	2	562	99.93%	99.77%	0.35%	99.75%	99.85%
Classifieds	1955	1622	8	1	324	99.94%	99.51%	0.31%	99.54%	99.72%
NewsPortal	1579	1257	4	1	317	99.92%	99.68%	0.31%	99.68%	99.80%
JobPortal	2912	2567	8	1	336	99.96%	99.69%	0.30%	99.69%	99.83%

Table 9: Comparison of SQLiGoT with existing techniques

Aspect →	1	2	3	4	5	6	7
SQLRand	Yes	Yes	No	No	No	Low	Low
SQL-DOM	Yes	Yes	Yes	No	No	High	Low
AMNESIA	Yes	Yes	Yes	Yes	No	Low	High

SQLProb	No	No	No	Yes	No	High	Medium
CANDID	No	Yes	Yes	Yes	No	Medium	Medium
Swaddler	No	No	Yes	Yes	No	High	Low
SQLiGoT	No	No	No	No	Yes	Low	High

Table 10: Types of SQL injection attacks detected

Type of Attack	SQLRand	SQL-DOM	AMNESIA	SQLProb	CANDID	Swaddler	SQLiGoT
Tautological attacks	•	•	•	•	•	○	○
Logically incorrect queries	×	•	•	•	○	○	•
UNION based attacks	•	•	•	•	•	○	•
Piggy-backed queries	•	•	•	•	•	○	•
Stored procedure attacks	×	×	×	•	○	○	•
Blind injection attacks	•	•	•	•	•	○	•
Time-based blind attacks	•	•	•	•	•	○	•
Alternate encodings	×	•	•	○	○	○	•

Legend: •= yes, ○ = partially, × = no