Profiling Database Applications to Detect SQL Injection Attacks

Elisa Bertino  
Purdue University  
bertino@cs.purdue.edu

Ashish Kamra  
Purdue University  
akamra@ecn.purdue.edu

James P. Early  
Purdue University  
earlyjp@purdue.edu

Abstract

Countering threats to an organization's internal databases from database applications is an important area of research. In this paper, we propose a novel framework based on anomaly detection techniques, to detect malicious behavior of database application programs. Specifically, we develop a fingerprint of an application program based on SQL queries submitted by it to a database. We then use association rule mining techniques on this fingerprint to extract useful rules. These rules succinctly represent the normal behavior of the database application. We then apply an anomaly detection algorithm to detect queries that do not conform to these rules. We further demonstrate how this model can be used to detect SQL Injection attacks on databases. We show the validity and usefulness of our approach on synthetically generated datasets and SQL injected queries. Experimental results show that our techniques are effective in addressing various types of SQL Injection threat scenarios.

1 Introduction

With rapid advancement of information technologies in the recent past, the amount of data stored within an organization's databases has increased manifold. A lot of these data is sensitive, and critical to an organization's day to day operations. Therefore, database vendors have traditionally strived to strengthen the security mechanisms surrounding these databases. One such approach is to install the databases behind an internal firewall, and restrict access to them through only application programs. To access a database, users connect to one such application, and the application submits queries to the database on behalf of the users. The threat to a database arises when these application programs do not behave properly and start behaving maliciously. Such behavior may often depend on the input provided by users, as is the case of the well known SQL Injection attack. As an example of such an attack, consider the following scenario of a database application program that tries to authenticate a user. The application presents the user with an authentication form and the user submits his username and password to the application. The application, at the server side, collects this information, creates a SQL query and submits it to the organization’s database in order to obtain the verification results. Consider the following application code fragment:

SQLQuery = "SELECT Username FROM Users WHERE Username = '" & strUsername & "' AND Password = '" & strPassword

In this code, the application receives the input supplied by the user through a form, embeds it directly into a SQL query and submits it to the database. Everything is fine as long as the user provides just the username and password. However, suppose that an attacker submits a login and password that looks like the following:

Login: ' OR ' = ' Password: ' OR ' = '

Such input would result in the following SQL query:

SELECT Username FROM Users WHERE Username = ' OR ' = ' AND Password = ' OR ' = '

Instead of comparing the user-supplied data with entries in the Users table, the query compares " (empty string) to " (empty string). This will always return a True result, and the attacker will then be logged in as the first user returned by the query. The above is a classical example of a SQL Injection attack. Essentially, SQL Injection is an attack exploiting applications that construct SQL statements from user-supplied input. When an application fails to properly validate user-supplied input, it is possible for an attacker to alter the construction of backend SQL statements. Several threat scenarios may arise because of these altered SQL queries. As we saw from the above example, it may allow an attacker to get unauthorized access to the database. Moreover, the modified SQL queries are executed with the privileges of the application program. An attacker may thus abuse the
privileges of the program in a manner unintended by the
application program designers.

The goal of our work is to develop an approach able to
detect malicious behavior of database applications. The key
idea of our approach is as follows. We create profiles of
application programs that can succinctly represent their nor-
mal behavior in terms of SQL queries submitted by these
programs to the database. Query traces from database logs
can be used for this purpose. We then use an anomaly detec-
tion model based on data mining techniques to detect behav-
or deviating from normal. The anomalous behavior that we
focus on in this paper is behavior related to SQL Injection
attacks. SQL Injection, traditionally, has been consid-
ered as an application level vulnerability and solutions have been
proposed at that level [8]. However, even though the mech-
nism of a SQL Injection attack is through an application,
the resource the security of which is directly threatened is
the database. Therefore, we propose a solution to enhance
the traditional security mechanisms of a Database Manage-
ment System (DBMS) to protect against such attacks. The
essential idea motivating our approach is that SQL Injection
can be modeled as an anomaly detection problem. In a typi-
cal database application, the input supplied by the users is
used to construct the where clause of queries at run time. An
SQL Injection attack typically involves malicious modifica-
tions to this input either by adding additional clauses or by
changing the structure of an existing clause. The projection
clause of the query, however, is not modified and remains
static because it is not constructed at run time. Driven by
this observation, we use association rule mining techniques
to derive rules that represent the associative relationships
among the various query attributes. We derive two sets of
rules specific to our task. The first set consists of rules bind-
ing the projection attributes of the query to the attributes
used in the where clause. The second set of rules represent
the relationship among the attributes in the where clause
of the query. These two sets of rules together form the profile
of an application. To detect an attack query, we check if the
relationship among its query attributes can be inferred by
the set of rules in the application profile. If not, our system
raises an alarm. We discuss the details of this methodology
in later sections.

In summary, our contributions in this paper are as fol-
low:

- We propose novel encoding schemes for SQL queries
to extract useful information from them.
- We present an approach for fingerprinting database ap-
lications based on the SQL queries submitted by them
to a database.
- We then present an anomaly detection model to detect
anomalous behavior of database applications.
- We further demonstrate how this model can be used to
detect SQL Injection at the database level.

1.1 Prior work

Anomaly detection models have been commonly used
in the field of network and host intrusion detection [5].
However, to date, there have been few approaches that ap-
ply anomaly detection techniques to identify anomalous
database application behavior. One such technique is by
Valeur et al. [18]. It builds a number of different statistical
query models using a set of typical application queries, and
then intercepts the new queries submitted to the databases
to check for anomalous behavior. Our work is similar in spirit
to their work but uses association rule mining techniques
to build application query profiles. Another work by Lee et
al. [14] considers the use of learning techniques for identi-
fying web-based attacks on databases. However, their meth-
ods for learning query fingerprints are fundamentally dif-
frent from ours. Apart from the above two learning based
techniques, there have been other approaches proposed in the
literature that use application code analysis. We direct the
reader to [8] for an in-depth survey of these approaches.

1.2 Paper road map

The rest of the paper is as follows. In section 2 we dis-
cuss our approach for encoding SQL queries and formulat-
ing application profiles. Section 3 discusses the rule min-
ing framework and the detection methodology. We report
experimental results on our approach in section 4. We con-
clude in section 5 with ideas for future work.

2 Application Profiles

The basic building block of an anomaly detection model
is a set of training records that can represent the normal be-
havior of an application. The training data that we consider
for our model are SQL queries submitted by a database ap-
lication to the database. The source for obtaining these
query traces is the database log. Once the raw training data
has been identified, the first step is to preprocess them to ex-
tract relevant features. In this section, we discuss the feature
extraction process from the SQL queries, and subsequent
formulation of an application fingerprint scheme based on
those features. The essential idea here is to efficiently en-
code the raw SQL queries so as to extract useful informa-
tion from them. In what follows, we discuss the encodings
schemes that we use to extract information from the queries.
2.1 Query encoding schemes

We devise two types of encodings to extract information from the queries. The first encoding, referred to as query encoding, captures the overall structure of a query in terms of the SQL command used, its projection attributes and its selection attributes. We use a bit-vector representation to indicate the presence and absence of a database attribute. The last attribute in the encoding, however, is an integer and it represents the total number of predicates present in the where clause of the query. The following example illustrates our approach. Assume a database $D$ in which the set of attributes (columns) $\mathcal{A} = A_1, A_2, \ldots, A_n$ have a lexicographic order imposed on them such that $A_i \preceq A_j$ for all $i < j$. Let $P, A_i$ represent a projection clause attribute and $S, A_j$ represent a selection clause attribute. Note the use of prefix $P$ to represent a projection attribute and $S$ to represent a selection attribute. Then a query $Q$ is encoded according to the following vector representation:

$$\text{SELECT UPDATE DELETE } P, A_1, P, A_2, \ldots, P, A_n$$
$$S, A_1, S, A_2, \ldots, S, A_n \text{ NUM-PRED}$$

Here, SELECT UPDATE DELETE $P, A_1, P, A_2, \ldots, P, A_n$ $S, A_1, S, A_2, \ldots, S, A_n$ are represented as bits, where 1 indicates the presence of that attribute in $Q$, and 0 indicates its absence. NUM-PRED is a non-negative integer representing the number of predicates in the where clause of $Q$.

As an example, consider a database storing two tables, $T_1$ and $T_2$, with columns $(A_1, B_1, C_1)$ and columns $(A_2, B_2, C_2)$, respectively. Consider the following query:

$$Q = \text{SELECT } T_1, A_1, T_2, B_2 \text{ FROM } T_1, T_2$$
$$\text{WHERE } T_1, C_1 = T_2, C_2 \text{ and } T_2, B_2 > 100$$

The encoding of $Q$ is given in Figure 1. We now formally define a query encoding.

**Definition 1. (Query Encoding)** Consider a database in which the set of attributes (columns) $\mathcal{A} = (A_1, A_2, \ldots, A_n)$ have a lexicographic order imposed on them such that $A_i \preceq A_j$ for all $i < j$. Then, Query Encoding (QE) for a SQL Query $Q$ is a vector representation of the syntactic structure of $Q$. It is composed of 4 distinct vector sets ($\text{[SQL-CMD]}, \text{[PROJECT-ATTR]}, \text{[SELECT-ATTR]}, \text{[NUM-PRED]}$). The first vector set $\text{[SQL-CMD]}$ denotes the command used in $Q$ and is a bit vector consisting of 3 attributes (SELECT UPDATE DELETE). The second vector set $\text{[PROJECT-ATTR]}$ is a bit vector encoding of $\mathcal{A}$ for the projection clause of $Q$. The third vector set $\text{[SELECT-ATTR]}$ is a bit vector encoding of $\mathcal{A}$ for the selection clause of $Q$. Finally, $\text{[NUM-PRED]}$ is a non-negative integer representing the number of predicates in $Q$.

The query encoding scheme effectively captures the information about the command, the projection attributes and the selection attributes. However, there is still information in the query predicates, such as join attributes, that is not accounted for by this scheme. To extract such information, we complement the previous encoding scheme with another encoding scheme referred to as predicate encoding, which provides fine-grained information about the predicates in a query. The predicate encoding scheme also assumes a total lexicographic order on the database attributes. In this scheme, a predicate in the where clause of the query is encoded as a bit vector according to the following vector representation:

$$\text{LHS_CONSTANT LHS}_A_1, \text{LHS}_A_2, \ldots, \text{LHS}_A_n$$
$$\text{RHS}_A_1, \text{RHS}_A_2, \ldots, \text{RHS}_A_n \text{ RHS_CONSTANT}$$

In the above encoding, 1 represents the presence and 0 the absence of an attribute in a predicate. LHS_CONSTANT = 1 if the predicate contains a constant literal or an empty string in the Left Hand Side (LHS). The same is true for the RHS_CONSTANT, LHS_A_i = 1 if A_i is present in the (LHS) of the predicate. RHS_A_i = 1 if A_i attribute is present in the Right Hand Side (RHS) of the predicate. Figure 2 shows how this encoding scheme can be used to encode the two predicates in the where clause of our example query $Q$. Note that the predicate encoding for $T_2, C_2 = T_1, C_1$ is same as for $T_2, C_1 = T_2, C_2$ since we assume a total order on the database attributes. Thus, before encoding, the predicate clause of a raw SQL query is transformed such that attribute on the left hand side of the predicates precedes the right hand side attribute according the lexicographic order.

**Definition 2. (Predicate Encoding)** Consider a database in which the set of attributes (columns) $\mathcal{A} = (A_1, A_2, \ldots, A_n)$ have a lexicographic order imposed on them such that $A_i \preceq A_j$ for all $i < j$. Then, Predicate Encoding (PE) for a predicate $P$ of a query $Q$ is a vector representation of the syntactic structure of $P$. It is composed of 2 distinct vector sets ($\text{[LHS-ATTR]}, \text{[RHS-ATTR]}$). The first vector set $\text{[LHS-ATTR]}$ is a bit vector encoding of $\mathcal{A}$ for the
Left Hand Side (LHS) of $P$. It also contains an attribute, LHS-CONSTANT which is 1 if the LHS of $P$ contains a constant literal or an empty string. Similarly, the second vector set [RHS-ATTR] is a bit vector encoding of $A$ for the Right Hand Side (RHS) of $P$. It also contains an attribute, RHS-CONSTANT which is 1 if the RHS of $P$ contains a constant literal or an empty string.

2.2 Application fingerprint

With the query encoding formats in place, we describe how the fingerprint of an application is formulated from these encodings. The fingerprint of an application consists of a set of SQL queries submitted by the application to the database. With the encoding schemes presented here, the query traces can now be represented by two matrices - the Query Matrix and the Predicate Matrix. The Query Matrix contains the query encoding representations of the query traces and the Predicate Matrix contains the predicate encoding representation of the predicates present in those queries. These two matrices together form the fingerprint of a database application. Note that since a database schema is likely to contain a large number of attributes, these matrices can be sparse as any single query is expected to touch only a small subset of these attributes. Efficient schemes have been proposed for compressing sparse matrices, and then mining information from them. These schemes are generally based on the idea of finding compact representations through discovery of some dominant patterns in data [6, 13, 7]. One such scheme is the Proximius framework by Mehmet et al. for compressing high-dimensional discrete attribute datasets [15]. Proximius provides a non-orthogonal matrix decomposition technique for reducing large data sets into a much smaller set of representative patterns, on which traditional analysis algorithms (including association rule mining) can be applied with minimal loss of accuracy. Although, we currently do not apply their techniques for compressing the application fingerprint, it would useful to apply this compression when we incorporate our approach in a real DBMS.

3 Anomaly Detection Framework

In the previous section, we have shown how the behavior of database applications can be captured effectively in the form of a fingerprint that consists of a Query Matrix and a Predicate Matrix. We now describe how such fingerprint can be used for detecting anomalous behavior of application programs. The essential idea is as follows. We use data mining techniques to extract patterns from the fingerprint. The mining technique that we use is association rule mining. We use rule mining algorithms to discover rules that can succinctly represent the normal behavior of the programs. For every new query under detection, we observe if the relationship among the query attributes can be inferred by the existing rules. If not, our system raises an alarm. Note that the anomalous behavior that we focus on in this paper is the behavior of programs attacked by SQL Injection. Thus, the detection methodology that we formulate is designed specifically to detect such attacks. However, the detection framework itself is general enough to detect other types of anomalous behavior. We start with a primer on the association rule mining paradigm.

3.1 Association rule mining basics

Association rule mining algorithms were introduced by Agrawal et al. to discover association rules between items in a large database containing sales transaction data [1]. An example of such a rule might be that 80% of customers that purchase formal trousers and shirts also purchase a tie. Since its introduction, this data mining technique has received a great deal of attention, primarily due to its applicability to a wide range of problems [11]. In brief, an association rule is an expression $X \Rightarrow Y$, where $X$ and $Y$ are both sets of items, $X$ is called the antecedent of the rule $X \Rightarrow Y$, the consequent. Given a database of transactions, a rule $X \Rightarrow Y$ expresses that whenever a transaction contains $X$, it also contains $Y$ with some probability measure. We describe the model in a formal manner below:

Let $I = i_1, i_2, \ldots, i_m$ be a set of attributes. Let $D$ be a database of transactions, where each transaction $T$ is a set of attribute values such that $T \subseteq I$. We say that a transaction $T$ contains $X$ if $X \subseteq T$. Now, an association rule is defined as an inference of the form $X \Rightarrow Y$, where $X \subseteq T$, $Y \subseteq T$, and $X \cap Y = \phi$. Every rule is characterized by two probability measures, support and confidence. The support $s$ of the rule is defined as $\frac{|X \cap Y|}{|D|}$, which means that $s\%$ of transaction in $D$ contains $X \cup Y$. This can also be interpreted as the probability $P(X \cup Y)$. The confidence $c$ of the rule is then defined as $\frac{|X \cap Y|}{|X|}$, which means that $c\%$ of transactions in $D$ that contain $X$ also contain $Y$. The rule confidence can also be understood as the conditional probability $P(Y|X)$.

Starting with the work in Agrawal et al. [1], a large body of algorithms exist for finding association rules from very large sets of transaction data. Faster algorithms for mining association rules were proposed in [2], while a hash-based
algorithm was discussed in [9]. Generalized association rules were presented in [3]. Methods for mining quantitative association rules were established in [4]. The notion of closed frequent itemsets and maximal frequent itemsets was introduced in [16] and [20] respectively. Moreover, there are algorithms to mine dense databases [12]. These approaches are supplemented by algorithms for online mining of association rules [10] and incremental algorithms [17]. A general survey and comparison of these techniques exist in [11].

### 3.2 Detection methodology

We now describe how the framework of association rule mining can be applied to discover rules from the application fingerprint. There are two sets of rules that we mine from the fingerprint information. These two sets of rules together form the application profile. The first set includes Profile Query Rules that we mine from the Query Matrix. A Profile Query Rule expresses the associations among the attributes used in the projection clause and those in the selection clause of the query. Thus, the rules that we mine have the following format:

\[
P.A_i \Rightarrow S.A_j
\]

(1)

\[
P.A_i \Rightarrow \text{NUM\_PRED}
\]

(2)

where \(P.A_i\) represent the projection attributes; \(S.A_j\) represent the selection attributes; \(i\) may be equal to \(j\).

For discovering rules of this type according to the rule mining framework, we consider the header attributes of the Query Matrix as the set \(I = i_1, i_2, \ldots, i_m\) of attributes. A transaction \(T\) is then represented by the query encoding scheme of the raw SQL query. The intuition behind mining rules of these forms is as follows. SQL Injection is an attack technique in which the query predicate is modified at run time in a malicious manner to alter the results returned by the query. The projection attributes however are not modified as they are generally not constructed at run time. Thus, by capturing the normal behavior of the applications according to the profile query rules, we can address two types of threat scenarios. First, if an attacker adds an additional predicate to the query, the query will violate rules of Type 2. Second, if an attacker changes some of the attributes in the predicate (but keeps the number of clauses same), rules of Type 1 can be used to detect such violations.

Though effective in preventing certain kinds of SQL Injection attacks, Profile Query Rules, however, are not able to detect against threat scenarios in which an attacker only modifies the structure of the predicates. As an example, consider the following threat scenario. Suppose that an attacker adds a predicate to the query that always evaluates to true and, at the same time, deletes another predicate. These two modifications combined have the effect that the number of predicates in the query does not change. Moreover, the predicates that are left also satisfy the existing Profile Query Rules. Such attacks cannot be detected by profile query rules alone as they do not capture the precise relationship among the attributes in the \(\text{where} \) clause of the query. To address such threat scenarios, we formulate rules, referred to as Profile Predicate Rules, from the Predicate Matrix of the application fingerprint. Recall that the Predicate Matrix consists of encodings of predicates in the \(\text{where} \) clause of the query. The set of rules that we mine from the Predicate Matrix capture the precise relationship between the attributes on the LHS and those on RHS of the predicate. The Profile Predicate Rules are represented as follows:

\[
LHS.A_i \Rightarrow RHS.A_j
\]

(3)

\[
LHS.A_i \Rightarrow \text{RHS\_CONSTANT}
\]

(4)

Predicate rules of Type 3 reflect the join condition between two database attributes, while rules of Type 4 model the boolean expressions when an attribute is equated to a constant literal or an empty string. With these rules in place, the threat scenario, when an attacker adds a predicate that always evaluates to true, can be easily addressed.

In what follows, we describe the anomaly detection methodology using these rules. For every new query under observation, we break it down to reveal the relationship among its attributes. We call these relationships, Detection Rules. As with the rules mined from the query traces, we derive two types of detection rules from the query under detection. They are Query Detection Rules and Predicate Detection Rules. The Query Detection Rules imply that the selection attributes in the query can be inferred from the projection attributes, and the Predicate Detection Rules imply that the RHS of the query predicates can be inferred.
from the LHS. We explain these rules with the help of the following example. Suppose the new query under detection is

\[
Q' = \text{SELECT} \ T_1, B_1, T_2, C_2 \ \text{FROM} \ T_1, T_2 \\
\text{WHERE} \ T_1.A_1 = T_2.A_2 \text{ and } T_1.C_1 = 100
\]

The Query Detection Rules that are generated for \( Q' \) are

\[
\begin{align*}
P.T_1.B_1 & \Rightarrow S.T_1.A_1 \\
P.T_1.B_1 & \Rightarrow S.T_2.A_2 \\
P.T_1.B_1 & \Rightarrow \text{NUM_PRED} = 2 \\
P.T_2.C_2 & \Rightarrow S.T_1.A_1 \\
P.T_2.C_2 & \Rightarrow S.T_2.A_2 \\
P.T_2.C_2 & \Rightarrow \text{NUM_PRED} = 2
\end{align*}
\]

and the Predicate Detection Rules are

\[
\begin{align*}
\text{LHS}.T_1.A_1 & \Rightarrow \text{RHS}.T_2.A_2 \\
\text{LHS}.T_1.C_1 & \Rightarrow \text{RHS}_\text{CONSTANT}
\end{align*}
\]

We now describe the anomaly detection procedure using these Detection Rules. The pseudo code for the algorithm is shown in figure 3. The anomaly detection algorithm checks how many Detection Rules can be matched against the rules existing in the application profile. For a strict match, all Detection Rules need to be matched. However, we provide a detection threshold parameter, \( \alpha \), to control the fraction of Detection Rules that must be matched to pass the detection test. There are two tests that a query under observation must pass to be declared benign. First, the Detection Query Rules are checked against the Profile Query Rules. The threshold parameter for this test is called \( \alpha_q \). If the fraction of Detection Query Rules that match the Profile Query Rules is greater than \( \alpha_q \), the query passes the first test. Then the Detection Predicate Rules are checked against the Profile Predicate Rules. The threshold parameter for this test is called \( \alpha_p \). If the fraction of Detection Predicate Inferences that match the Detection Predicate Rules is greater than \( \alpha_p \), the query passes the second test and is then declared benign. If it fails either of the two tests, it is declared anomalous. Note that the detection threshold parameters \( \alpha_q \) and \( \alpha_p \) can be modified to control the false positive and false negative rates of the detection methodology. In the next section, we briefly discuss the association rule mining algorithm that we employ to discover the rules. We then report experimental results on our detection methodology.

4 Experimental Evaluation

4.1 Algorithm and implementation

Since the initial proposal of the association rule mining technique by Agrawal et al. [1], a number of algorithms and measures have been explored that improve upon the original algorithm. One such algorithm is the popular \textit{apriori} algorithm developed by Agrawal et al. [2] which we use for evaluating our approach. In what follows, we briefly discuss the \textit{apriori} algorithm. We refer the reader to the original paper by Agrawal [2] for details.

As with most association rule mining algorithms, \textit{apriori} first calculates frequent itemsets from a database of transactions. An itemset is a collection of items that is contained in a single transaction. An itemset is called frequent if its support value is above a user specified \textit{minimum-support} parameter. The \textit{apriori} algorithm reduces the search space of potential candidates for frequent itemsets by concluding \textit{a priori} that certain combinations of itemsets cannot possibly satisfy the \textit{minimum-support} requirement. After the frequent itemset generation phase, the discovery of rules is rather straightforward. For every frequent itemset \( L \), all non-empty subsets of \( L \) are found. For every such subset \( A \), a rule of the form \( A \Rightarrow (L - A) \) is returned, if it satisfies the user supplied \textit{minimum-confidence} parameter. In our implementation, we used the Java API of the popular Weka toolkit for machine learning algorithms [19]. However, we modified Weka's implementation of \textit{apriori} to address the specific requirements of our problem. Weka, as per the original \textit{apriori} algorithm [19], returns all rules that satisfy the \textit{minimum-support} and \textit{minimum-confidence} parameters. We require only the rules having certain attributes in the rule antecedent and consequent. We implemented a filtering scheme to satisfy such constraints. Finally, note that the choice of a specific rule mining algorithm may affect the time taken for the rule generation phase but it does not affect the general applicability of our methods.

4.2 Quality measures and dataset generation

We measure the false positives and false negatives generated by the detection algorithm to evaluate the quality of our approach. A \textit{false positive} is generated when a benign query is declared anomalous by the detection model. A \textit{false negative} is generated when an anomalous query is declared normal. Since SQL traces from a real database application were unavailable, we use synthetic datasets for the experiments. In what follows, we describe the methodology for generating synthetic query traces.

Training dataset generation: Assume a database of \( n \) tables \( T_1, T_2, \ldots, T_n \). Assume each table to contain
Function boolean detectionAlgo(detectionQueryInstance, profileQueryRules, profilePredicateRules, threshold) {
    detectionQueryRules = getDetectionQueryRules(detectionQueryInstance);
    detectionPredicateRules = getDetectionPredicateRules(detectionQueryInstance);
    fractionQueryMatches = getMatches(detectionQueryRules, profileQueryRules);
    fractionPredicateMatches = getMatches(detectionPredicateRules, profilePredicateRules);
    
    if (fractionQueryMatches < threshold_query)
        queryInstanceAnomalous = true;
    else if (fractionPredicateMatches < threshold_predicate)
        queryInstanceAnomalous = true;
    return queryInstanceAnomalous;
}

**Figure 3.** Pseudo code for the detection algorithm

$m$ attributes $A_1, A_2, \ldots, A_m$. Let $A_{ij}$ represent the $j$-attribute of $i$-table. We assume the application to be used by $k$ different users, each generating their own distinct query trace set. Thus, we get $k$ potentially overlapping query trace sets $S_1, S_2, \ldots, S_k$. Each set $S_i$ is assigned a probability $P(S_i)$ of generating queries. Moreover, each set $S_i$ is also assigned a probability $P(S_i, T_j)$ of projecting attributes from the $j$-table and a probability $P(S_i, A_{jk})$ of projecting the $k$-attribute of $j$-table. Similar probabilities are defined for the selection attributes. This probabilistic model generates all the flexibility to control the heterogeneity of the resulting training dataset.

Anomalous query traces generation: We generate two types of anomalies to reflect the SQL Injection threat scenarios. In the first type, we increase the number of predicates in the normal query by 1. This reflects the scenario when an attacker injects a malicious predicate that always evaluates to true. Note that such a predicate may lead to Predicate Detection Rules of the form

\[
LHS_{\text{CONSTANT}} \Rightarrow \text{RHS}_{\text{CONSTANT}} \lor \quad LHS_{\text{A}_i} \Rightarrow \text{RHS}_{\text{A}_i}
\]

Such rules are not expected to be satisfied by any of the Profile Predicate Rules. For the second type of anomalous queries, we modify the structure of the predicate clause of the normal queries so as to distort the normal associative relationship as represented by the Profile Predicate Rules. We discuss the experimental results on both types of anomalous queries in the next section.

**4.3 Results**

The first dataset (dataset 1) that we consider is of highly heterogenous nature, such that the number of distinct queries is very high. Intuitively, such heterogenous queries must be difficult to discriminate for the rule miner as the support of any combination of projection and selection attributes would be quite low. This is reflected in the high false positive rate of 45.9% for this dataset even with a $\min - \text{support}$ value as low as 0.01. Thus, we take this dataset as the baseline case to measure the performance of other less heterogenous datasets.

**Figure 6.** Profile Rules and min-support

The second dataset (dataset 2) that we consider is of less heterogenous nature as reflected by its query generation model in figure 4 and 5. This dataset is characterized by a database schema consisting of 50 attributes and 6 sets of query traces. Figures 6 and 9 show the results for this dataset. Here, Type 1 false negatives are those generated by Type 1 anomalous queries, while Type 2 false negatives are
generated by anomalous queries of Type 2. As expected, the number of Profile Query Rules and Profile Predicate Rules increase with decreasing values of min-support. With increasing number of rules, the false positive rate goes down as less benign queries are detected as anomalous. There are two observations from these results that are worth mentioning. First is the zero false negative rate at all levels of min-support. This shows the usefulness of our approach in detecting SQL Injection attack queries modeled as anomalies. The second observation is the low number of profile rules generated at all values of min-support. This makes the detection process highly efficient as evident in figure 8.

Figure 10 shows the effect of varying the detection threshold parameters ($\alpha_q$ and $\alpha_p$) on the false positive and false negative rates. For this experiment, we keep both $\alpha_q$ and $\alpha_p$ at the same level. By decreasing the threshold, we relax the criterion for declaring a query as anomalous. Thus we see that the false positive rate goes down and the consequently, the false negative rate goes up with decreasing values of detection threshold. In this way, the detection threshold parameters allows us to maintain a balance between the desired false positive and false negative rates.

Finally we discuss results related to the efficiency of our method. The detection time per query is shown in figure 8. Even the maximum value of 1 millisecond at the lowest min-support level of 0.01 is quite low. This confirms the low overhead associated with our approach. Figure 7 shows the variation in profile generation time for decreasing values of min-support for this dataset. Note that this time includes the time taken for formation of both Profile Query Rules and Profile Predicate Rules. The $\text{apriori}$ algorithm spends maximum amount of its time in generating the frequent itemsets. Time taken for the rule generation phase and subsequent filtering is negligible. Thus, the run time for the profile generation phase can be improved by considering other efficient ways of generating frequent itemsets such as [16, 20].

In summary, the experimental results showcase the usefulness of our approach in detecting SQL Injection attacks modeled as anomalous queries.
5 Conclusions and Future Work

In this paper, we presented an anomaly detection model to detect anomalous SQL queries submitted by database application programs to a database. As part of this, we modeled SQL injection attacks on databases as anomalous behavior that we try to detect. We considered SQL queries submitted by an application program to the database as training data for our detection model. We extracted useful features from these queries to represent the fingerprint of an application. We then used association rule mining techniques on this fingerprint to derive two sets of rules that make up the normal profile of the application. For every new query under detection, we checked if the relationship among its attributes can be inferred from the normal profile of the application. If yes, we raised an alarm. Experimental results on synthetic generated datasets showed the usefulness of our approach.

We plan to extend this work on the following lines. We will consider more efficient rule generation algorithms that may support datasets with large number of attributes. Since, query traces from real database application are difficult to obtain, we plan to build a SQL query generator that generates queries based on a standard data distribution model. Finally, for the system to be useful we must be able to incorporate it into a real DBMS. We plan to explore more efficient representations of an application fingerprint and profile for this purpose.

References


