Learning Fingerprints for a Database Intrusion Detection System

Sin Yeung Lee, Wai Lup Low, and Pei Yuen Wong

Computer Security Laboratory, DSO National Laboratories, Singapore
{lsinyeun,lwailup,wpeiyuen}@dso.org.sg

Abstract. There is a growing security concern on the increasing number of databases that are accessible through the Internet. Such databases may contain sensitive information like credit card numbers and personal medical histories. Many e-service providers are reported to be leaking customers' information through their websites. The hackers exploited poorly coded programs that interface with backend databases using SQL injection techniques. We developed an architectural framework, DIDAFIT (Detecting Intrusions in DAtabases through FIngerprinting Transactions) [1], that can efficiently detect illegitimate database accesses. The system works by matching SQL statements against a known set of legitimate database transaction fingerprints. In this paper, we explore the various issues that arise in the collation, representation and summarization of this potentially huge set of legitimate transaction fingerprints. We describe an algorithm that summarizes the raw transactional SQL queries into compact regular expressions. This representation can be used to match against incoming database transactions efficiently. A set of heuristics is used during the summarization process to ensure that the level of false negatives remains low. This algorithm also takes into consideration incomplete logs and heuristically identifies "*high risk*" transactions.

1 Introduction

Nowadays, most e-commerce sites offering some kind of online services have a database at its backend. Applications accessing these databases support a large variety of activities. Data found in these databases ranges from personal information and banking transactions to medical records and commercial secrets. Any breach of security to these databases can result in tarnished reputation for the organization, loss of customers' confidence and might even result in lawsuits. Unfortunately, recent reports indicate that there is a large increase in the number of security breaches, which resulted in theft of transaction information and financial fraud [2, 3, 4]. Clearly, it is important that the data in these databases be protected from unauthorized access and modification.

One mechanism to safeguard the information in these databases is to use intrusion detection systems (IDS). These systems aim to detect intrusions as early as possible, so that any damage caused by the intrusions is minimized.

D. Gollmann et al. (Eds.): ESORICS 2002, LNCS 2502, pp. 264–279, 2002.

They function as sentinels and ensure that any compromise on the integrity and confidentiality of the data is detected and reported as soon as possible. Intrusion detection research is not new and has been on going for many years. However, previous efforts were focused on *network-based intrusion detection* and *host-based intrusion detection*. Network-based intrusion detection typically works by monitoring network traffic and host-based intrusion detection works by monitoring log files in the hosts. Both network- and host-based intrusion detection systems look for attack signatures, which are specific patterns that usually indicate malicious or suspicious intent, to identify intrusions. There are numerous commercial network- and host-based intrusion detection systems in the market today, and the market leaders include RealSecure [5], NFR [6], Dragon [7], Cisco [8] and Symantec [9]. There are also IDS that are free and highly acclaimed (e.g. Snort [10]).

However, these intrusion detection systems do not work at the application layer, which can potentially offer more accurate and precise detection for the targeted application. The distinctive characteristics of database management systems (DBMSes), together with their widespread use and the invaluable data they hold, make it vital to detect any intrusion attempts made at the databases. Therefore, intrusion detection models and techniques specially designed for databases are becoming imperative needs. There are recent reports [4] in which SQL injection techniques, which refer to the use of carefully crafted and malicious SQL statements, were used by the intruders to pilfer sensitive information. SQL injection will be further discussed in a later section. This reinforces the point that intrusion detection systems should not only be employed at the network and hosts, but also at the database systems where the critical information assets lie.

DIDAFIT (Detecting Intrusions in DAtabases through FIngerprinting Transactions) is a system developed to perform database intrusion detection at the application level. It works by fingerprinting access patterns of legitimate database transactions, and using them to identify database intrusions. The framework for DIDAFIT has been described in [1]. This paper describes how the fingerprints for database transactions can be represented and presents an algorithm to learn and summarize SQL statements into fingerprints. The main contribution of this work is a technique to efficiently summarize SQL statements queries into compact and effective regular expression fingerprints. The technique can handle incomplete training data sets and uses heuristics to maintain false negatives (missed attacks) at low levels.

The rest of the paper is as follows. Section 2 discusses two basic concepts necessary for the understanding of subsequent sections. A brief introduction to the DIDAFIT framework is given in Section 3. The issues that arise during fingerprint learning and derivation are discussed in Section 4. In Section 5, our algorithm for fingerprint learning is detailed together with a full example. We survey the related works in Section 6, and conclude in Section 7.

2 Preliminaries

In this section, we introduce *SQL injection* and *SQL fingerprints* which are necessary for understanding the rest of the paper.

2.1 SQL Injection

Application users do not usually communicate with the database server directly, but through the application server. Although there is no direct interaction with the database server, it is possible that unauthorized users can access the database in ways unintended by the developer. This is made possible by carelessly designed applications, database server holes, as well as application server exploits. One technique of exploiting carelessly written database applications is *SQL injection* [11, 12, 13]. SQL injection refers to crafting SQL statements using "string building" techniques to trick the application server into executing the intruder's (often malicious) code. Possible results of actions by the injected code include information disclosure, unauthorised data modification, deletion of database or even escalation of the intruder's database privileges to that of the administrator's.

As an illustration, consider the following Perl script:

```
...
my $passwd = $cgi->param('passwd');
my $name= $cgi->param('name');
$sql = "select * from cust where name='$name'".
       " and passwd='$passwd'";
$sth = $dbh->prepare($sql);
$sth->execute;
if (!($sth->fetch)) {
   report_illegal_user();
} ...
```

The script shown is a typical procedure for login checking. It is supposed to verify if the user has supplied the user name and his/her password correctly. However, this script is vulnerable to SQL injection attacks. A malicious user can enter the following text into the password field of the submitted form:

```
x' OR 'x'='x
```

Assuming the user name is "alice" and the password is as above, the prepared SQL statement becomes

```
select * from customer
where name='alice' and passwd='x'
      OR 'x'='x'
```

The `where` clause of this statement will always be true since the intruder has carefully injected a " `OR 'x'='x'` " clause into it. This makes the result set of the query non-empty no matter what password is supplied. The malicious user can now log in as the user “alice” without knowing the password. The main reason for this vulnerability is the carelessly written procedure. DIDAFIT can detect such attacks on the database and other anomalous data accesses. Conventional solutions to this vulnerability include the use of stored procedures and checking user parameters before using them (e.g. dangerous characters such as ’ should not be allowed in the input). While stored procedures tend to be non-portable and require longer development time, methods to enforce good programming practices are beyond the scope of this work.

2.2 SQL Fingerprints

For most applications with database services, the SQL statements submitted to the database are typically generated by some server-side scripts/programs. These statements are generated in a predictable manner and this regularity gives rise to opportunities to characterize valid transactions with some sort of signatures.

For example, assume we have a `delete order` transaction that deletes from the `order` table. Further assume that an order can only be deleted by specifying its `orderID`. A typical **valid** SQL statement can be

```
delete from order where orderID=’12573’;
```

Now, suppose the database server receives the following SQL statement:

```
delete from order where custid=’eric’;
```

This SQL statement does not conform to the pattern of the SQL statements for `delete order` transaction (i.e. it uses `custid` as the criteria for filtering instead of `orderID`). This statement could be the result of an intruder.

We have presented the idea behind fingerprinting database transactions. DIDAFIT uses a fingerprinting technique that derives signatures (which we call fingerprints) to help identify illegitimate SQL statements. In DIDAFIT, each fingerprint characterizes one set of SQL statements. Our fingerprints (as described in [1]) uses regular expressions. A single fingerprint can precisely represent various variants of SQL statements that can result from a transaction.

For example, we have a transaction that can query the order table given a customer ID (`custid`) and a range of values for the order amount (`amt`). The code below shows how this service may be coded in Perl.

```
$sqlstm = "SELECT orderID, amt from order where ";
$sqlstm .= "custid=’$custid’ and ";
$sqlstm .= "amt>$min_amt and amt<$max_amt";
```

We list below some possible SQL statements generated from this code:

```
SELECT orderID, amt from order where custid=’3822’ and amt>20 and amt<100
SELECT orderID, amt from order where custid=’7312’ and amt>10 and amt<200
```

All variants of queries generated by the code above can be represented using this regular expression:

```
^SELECT orderID, amt from order where custid='[^']*'
        and amt>[[:digit:]]+ and amt<[[:digit:]]+$
```

Notably,

1. The literal containing the value of `custid` has been replaced with the regular expression `'[^']*'`, which represents a quoted string.
2. The numerical value of `amt` is represented by the regular expression `[[:digit:]]+`, which represents an integer.
3. Finally, the whole expression begins with a `^` and ends with a `$` to prevent any additional clauses from being injected into the statement (such as using the `union` clause to append another query).

3 DIDAFIT: An Overview

DIDAFIT is designed as a misuse detection system. It can be broadly classified as a signature-based IDS with enhanced capabilities for learning and deducing new signatures. The overall architecture for DIDAFIT is shown in Figure 1. The meaning of the numbered flows in Figure 1 are as follows:

1. The application user issues a service request to the application server. This transaction may or may not be legitimate.
2. The application server formulates the necessary SQL statements and issues them to the database server through the database user.
3. The database user logs into the database. The database session is traced and the SQL statements received from the application are channeled to the misuse detection module.
4. In the misuse detection module, the received SQL statements are matched with the set of fingerprints of legitimate database transactions.
5. Anomalies or intrusions are then channeled to the reaction modules for the appropriate action to be taken. Actions that can be taken include alerting the administrators, sounding the alarm on the console and paging the duty personnel.
6. Output is returned to the application user (if applicable).

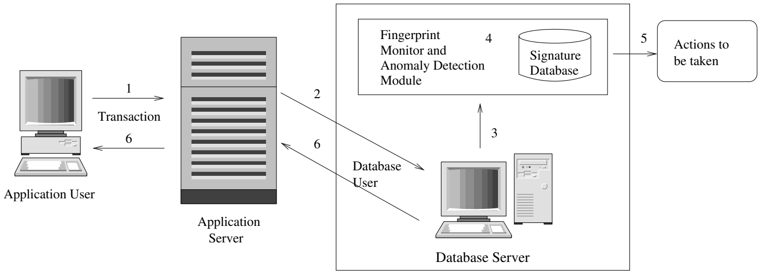

Fig. 1. Architecture for DIDAFIT

DIDAFIT involves three components:

1. An utility to log all the SQL statements submitted to the database server.
   We need to capture the submitted SQL statements for the database transactions in order to compare them with those in the "legitimate" fingerprint database. Different database systems offer different ways to capture the SQL statements. For example, Oracle provides the `sql_trace` [14] utility that can be used to trace all database operations in a database session of an user. The trace results are logged to a file, but we can channel the data to a monitoring and misuse detection module. Note that the `sql_trace` utility is not designed for this purpose. Rather, its intended use was to aid in the process of optimizing database performance. However, we make use of its capability to log SQL statements executed by the database engine. A concern with using the trace facility of database systems is its impact on the performance of the databases. Our experiments in [1] show that the impact on performance is extremely small.
2. A process to derive the fingerprints for SQL statements of legitimate database transactions.
   We propose the use of regular expressions to represent the derived fingerprints as described in Section 2.2.
3. A database of "legitimate" fingerprints to be used for database intrusion detection.
   To make use of the fingerprints, every submitted SQL statement is matched with the set of legitimate fingerprints. If the SQL statement cannot match any of the fingerprints, then an intrusion may have occurred.
   As an illustration, consider the example of `order` table queries (Section 2.2). Suppose our database server receives the following SQL statement.
   ```
   select orderID, amt from order;
   ```
   This statement does not match our legitimate fingerprint shown previously. This is caused by the missing `custid` input, which is mandatory by our signature. Hence, this anomalous statement is detected.
   There are many software packages and utilities that can process regular expressions. The standard Unix tool `egrep` is one such software that can handle the matching of our fingerprints efficiently.

4 Issues in Automated Fingerprints Learning and Derivation

One obvious method of generating the complete set of fingerprints for all database transactions is to do a code-walkthrough. However, this may not be feasible for several reasons. The code may contain sensitive business logic and not available for the walkthrough for some applications. The code base for many applications are also large and changes frequently. Maintaining the consistency and keeping the set of finerprints updated will require a lot of effort. Hence, it is important that the fingerprints be automatically learnt and deduced as much as possible.

In this section, we first discuss the problems and challenges that arise when we automate the learning and derivation of the fingerprints for the database transactions. We propose possible solutions after that.

4.1 Problems and Challenges

An obvious method to "learn" fingerprints is to use each legitimate SQL statement in the training log as a fingerprint. However, it suffers from the following problems:

1. *The set of SQL statements collected is a very large set.* If each unprocessed SQL statement forms a fingerprint, the size of the fingerprint database becomes unmanageable. We need a way to automatically summarize the SQL statements. Consider the following SQL statements:
   ```
   delete from order where orderID='13245';
   delete from order where orderID='45718';
   ```
   Although the `orderID`s are different, they serve the same function, which is to remove a particular order based on the `orderID` and hence, can be summarized to a single fingerprint:
   ```
   delete from order where orderID=#$ORDERID;
   ```
   where `#$ORDERID` is a meta-constant to denote a string literal in the domain of `orderID`. However, we need to avoid over-summarization. For example,
   ```
   delete from order where orderType='UNDELIVERABLE';
   delete from order where orderType='NORMAL';
   ```
   should not be summarized to a single fingerprint. The first SQL statement may be a typical statement used during maintenance. However, it is possible that the second statement originated from a malicious attack (why should anyone delete all "normal" orders?). Over-summarization may cause DIDAFIT to miss such illegitimate statements. Thus, our learning algorithm must be able to group similar SQL statements into fingerprints, but yet does not increase the probability of missing attacks.
2. *The SQL transactions in the training trace log may contain illegitimate statements from past intrusion activities.* We do not assume that all the SQL statements in the training log to be legitimate. Our learning algorithm must be able to detect potentially invalid SQL statements, even within the training set of SQL statements.
3. *The trace log may not be complete.* Some legitimate transactions may not be executed during the period of monitoring. With incomplete input for training, DIDAFIT may treat unseen, but legitimate statements as illegitimate. This may give rise to a large number of false positives. Thus, our learning algorithm should be able to deduce a set of possibly legitimate fingerprints with an associated level of confidence, that are missing from the training data. The legitimacy of these deduced fingerprints can be ascertained by the DBA.

```
L1   select orderID, Amt from order where custID=’2317’;
L2   select orderID, Amt from order where custID=’1920’;
L3   select orderID from order where custID=’3571’ and amt>100 and amt<300;
L4   select prod_desc from product where prodID=’X3175’;
L5   select orderID from order where custID=’’ or TRUE or custID=’’;
L6   select comment from prod_comment where prodID=’X3175’ and conf=’PUBLIC’;
L7   select orderID, Amt from order where custID=’8256’;
L8   select orderID from order where custID=’1354’ and odate=’1-Jan-1999’ and amt>500;
L9   select comment from prod_comment where prodID=’X0507’ and conf=’PRIVATE’;
L10  select prod_desc from product where prodID=’X1754’;
L11  select orderID from order where custID=’1028’ and odate=’1-Jan-1999’ and amt<1000;
L12  select orderID, Amt from order where custID=’1754’;
L13  select comment from prod_comment where prodID=’X1754’ and conf=’PUBLIC’;
L14  select comment from prod_comment where prodID=’X0507’ and conf=’PUBLIC’;
L15  select comment from prod_comment where prodID=’X3075’ and conf=’PUBLIC’;
L16  select prod_desc from product where prodID=’X0675’;
L17  select orderID, Amt from order where custID=’8317’;
```

Fig. 2. A snapshot of the SQL trace log for an E-mall

Before we present our solutions to these three challenges, we consider an online E-mall application with the following business rules:

- Users can create and delete their own orders.
- Users can perform searches on their own orders and limit the search by specifying the order ID (`orderID`), product ID (`prodID`), the range of the order's value (`amt`), and/or the date of order (`odate`).
- Users can see the description of the products (`prod_desc`).
- Users can only see the public comments on the product (`conf=’PUBLIC’`), but not internal comments, which are reserved only for the staff of the E-mall.

A snapshot of the SQL trace log is shown in Figure 2. This example will be used in the examples for the rest of this work.

4.2 Selective Summarization of Literals

Observe from the logs in Figure 2 that L1 differs from L2 only by the value of the `custID` parameter. They query the same attributes for a given customer ID. Our learning algorithm can group these SQL statements under the same fingerprint. This can be done by first replacing each literal by a meta-constant. In this case, each string literal that represents the value of a `custID` attribute is replaced by the token `#$CUSTID`. The summarized fingerprint for L1 and L2 is

```
F1   select orderID, Amt from order where custID=#$CUSTID;
```

Likewise, the next three log records (L3,L4,L5) give the following three fingerprints:

```
F2   select orderID from order where custID=#$CUSTID and amt>#%AMT and amt<#%AMT;
F3   select prod_desc from product where prodID=#$PRODID;
F4   select orderID from order where custID=#$CUSTID or TRUE or custID=#$CUSTID;
```

However, as mentioned previously, replacing all literals to wild-card tokens indiscriminately may cause some malicious SQL statements to go undetected. Consider L6, L9, L13, L14 and L15 , the first attribute in the WHERE-clause (product ID `prodID`), can take any value, but the second attribute, (confidentiality `conf`), takes only the value "`PUBLIC`". This corresponds well to the "business rule" that customers can see only the public comments but not otherwise. Thus, we should only replace the literals of the `prodID` with the wild-card token and let the literals of the `conf` attribute remain unchanged, i.e.

```
select comment from prod_comment
  where prodID=#$PRODID and conf='PUBLIC';
```

The difference between the two treatments lies on the fact that `prodID` is not from a small set of pre-specified values. This value carries no other implication, except to identify a product. On the other hand, the valid values of `conf` is restricted to a **small** list of pre-determined values. Values such as "PUBLIC", "PRIVATE" and "SECRET" are not only used to identify tuples in the database, but also carry implications for the operations and sensitivity of the tuples. Hence, it is not advisable to summarize these literals into one token.

In view of this, our algorithm will replace a literal with a token only when the literal corresponds to a domain that carries no implicit meaning for operations and data sensitivity. Such domains can be determined by consulting the DBA. However, in the event that this information is unavailable, we can assume that if the range of values in the domain is very large (or unbounded), then it is unlikely that such value has an implicit meaning. To test if the domain is unbounded, we test the growth rate of the number of unique values for the attribute as the size of the sample increases. If the domain is unbounded, the growth rate will be constant. That is to say:

$$\text{Number of distinct literals found} \propto s$$

where s is the number of the samples in the observation. On the other hand, if the domain contains only a few fixed constants, then the number of distinct values will be similar to the standard diminishing growth curve,

$$\text{Number of distinct literals} = N_0(1 - e^{-\lambda s})$$

This behavior can be easily tested with many statistical methods such as the non-parametric one-sample Komoglov-Smirnov's D Statistics test (KS-test). For example, consider the set of trace L6, L9, L13, L14, L15 from Figure 2. The `prodID` attribute employs four literals: `'X3175'`, `'X0507'`, `'X1754'` and `'X3075'` in the first conjunction in the WHERE-clause.

Using the one-sample KS-test, the D-statistic for testing the linear growth can be computed as follows:

$$D = \sqrt{5}\max\{|\frac{1}{4} - \frac{1}{5}|, |\frac{2}{4} - \frac{2}{5}|, |\frac{3}{4} - \frac{3}{5}|, |\frac{3}{4} - \frac{4}{5}|\} = 0.15$$

Table 1. Distribution of the `prodID` Literal

Number of statements sampled	1 (L6)	2 (L6,L9)	3 (L6,L9,L13)	4 (L6,L9,L13,L14)	5 (L6,L9,L13,L14,L15)
Total no. of unique literals observed	1	2	3	3	4
Cumulative distribution	1/4	2/4	3/4	3/4	4/4
Expected uniform distribution	1/5	2/5	3/5	4/5	5/5

At α=90% significance level, this value should be at most 0.563. Thus, we accept the hypothesis that the domain of `prodID` is unbounded.

On the other hand, the `conf` attribute has 2 unique literals in the 5-statement sample: `'PUBLIC'` and `'PRIVATE'`. The D-statistic is computed likewise as:

$$D = \sqrt{5}\max\{|\frac{1}{2} - \frac{1}{5}|, |\frac{2}{2} - \frac{2}{5}|, |\frac{2}{2} - \frac{3}{5}|, |\frac{2}{2} - \frac{4}{5}|\} = 0.6$$

It exceeds 0.563. Thus, we reject the hypothesis that the domain of `conf` is unbounded. Hence, we should retain all the constants related with `conf` when generating the fingerprints.

4.3 Detection of High-Risk Transactions

It is obvious that L5 is an instance of SQL injection. It is caused by injecting the string "`' or TRUE or custID='`" into the `custID` parameter. We consider this fingerprint to be *rare*. A fingerprint is *rare* if its frequency is statistically below a small threshold. The occurrence of a rare fingerprint warns of a possible malicious SQL statement.

In our algorithm, we tabulate the frequency of each fingerprint learnt. If a fingerprint occurs frequently, then it is safe to include it in the fingerprint database. For instance, consider the four fingerprints, F1, F2, F3 and F4 (in Section 4.2).

Fingerprint F1 matches 5 statements in the trace of size 17 (Figure 2). A z-test can conclude that the frequency is not close to 0. Similarly, fingerprint F3 can be considered safe based on its frequency. On the other hand, fingerprints F2 and F4 matches only one statement each in the trace. Statistically, at 95% confidence level, we cannot deny that their frequency is 0. In other words, they may correspond to potentially malicious statements and require further investigation.

To further refine our decision to ascertain the legitimacy of F2, we observe that F2 differs from F1 as follows:

1. F2 has a more restrictive WHERE-clause compared to F1. It specifies more conditions using the "AND" operator in the WHERE-clause than F1.
2. F2 selects fewer attributes than F1.

Thus, we can consider F2 to be safe because for any SQL statement that matches F2, the set of returned tuples is a subset of the tuples returned by some SQL statement matching F1 (which is a legitimate fingerprint). Thus, even if some SQL statements that match F2 is malicious, it cannot reveal more information than some legitimate query that is allowed by F1.

On the other hand, F4 cannot be considered safe. F4's WHERE-conditions are joined using the "OR" operator. Thus, it is possible that a query that matches F4 can reveal more tuples than what is allowed by the legitimate fingerprints.

Thus, our learning algorithm decides that an rare fingerprint $\mathcal{F}$ is legitimate if there exists another legitimate fingerprint $\mathcal{F}'$, such that $\mathcal{F}$ differs from $\mathcal{F}'$ only by

1. any extra conditions in the WHERE-clause of $\mathcal{F}$ that is missing from $\mathcal{F}'$ is joined with the "AND" operator; and
2. $\mathcal{F}$ selects an equal number of or fewer columns than $\mathcal{F}'$.

4.4 Deduction of Missing Fingerprints.

As mentioned previously, some legitimate fingerprints might not appear in the trace log due to incomplete training data. This incompleteness can cause many false alarms during the actual operation. To reduce this problem, we propose an algorithm to deduce a set of possible missing fingerprints. This set of missing fingerprints will then be presented to the DBA for confirmation to be included in the legitimate fingerprint set.

Consider the transaction set L3, L8, L11 of the log in Figure 2. These logs can be captured by the following three fingerprints:

```
P1   select orderID from order where custID=#$CUSTID and amt>#%AMT and amt<#%AMT;
P2   select orderID from order where custID=#$CUSTID and odate=#$ODATE and amt>#%AMT;
P3   select orderID from order where custID=#$CUSTID and odate=#$ODATE and amt<#%AMT;
```

The three fingerprints share the following properties:

1. Except for the conditions at the WHERE-clause, they are identical.
2. The WHERE-clause of each pattern uses three out of the following four conjuncts:
   "`custID=#$CUSTID`", "`odate=#$ODATE`", "`amt>#%AMT`" and "`amt<#%AMT`".

It is possible that there is another fingerprint, that uses all four conjuncts:

```
P4   select orderID from order where custID=#$CUSTID and odate=$#ODATE
         and amt>$%AMT and amt<$%AMT;
```

P4 compacts all the nine conjuncts used in P1, P2 and P3 into only four conjuncts. Indeed, this scenario can be generated by the following Perl script:

```
$sqlstm = "select orderID from order where custID='$custID'";
if !($odate eq "") {
  $sqlstm .= " and odate='$odate'";
}
if ($min_amt > 0) {
  $sqlstm .= " and amt>$min_amt";
}
if ($max_amt > 0) {
  $sqlstm .= " and amt<$max_amt";
}
```

In general, a fingerprint $\mathcal{F}$ is a *derived fingerprint* from fingerprints $\mathcal{F}_1, \ldots, \mathcal{F}_m$ with compactness η if

1. Each pair of fingerprints $\mathcal{F}_i, \mathcal{F}_j$ $(1 \leq i, j \leq m)$, as well as $\mathcal{F}, \mathcal{F}_i$ $(1 \leq i \leq m)$, differs only by some missing conjuncts in their WHERE-clauses.
2. The condition in the WHERE-clause of $\mathcal{F}$ subsumes every condition in the WHERE-clause of each $\mathcal{F}_j$ $(1 \leq j \leq m)$.
3. Each conjunct in the WHERE-clause of $\mathcal{F}$ appears in at least the WHERE-clause of at least one $\mathcal{F}_j$ $(1 \leq j \leq m)$.
4. Furthermore, η $(\eta \geq 1)$ is the ratio of the total number of possibly identical conjuncts in $\mathcal{F}_1, \ldots, \mathcal{F}_m$ over the number of conjuncts in $\mathcal{F}$.

In our derived fingerprint P4, the number of conjuncts used in the WHERE-clause is 4. The total number of conjuncts used in the three fingerprints (P1,P2,P3) is 9. Hence, P4's compactness is $\frac{9}{4}$. High compactness gives higher confidence that the fingerprints $\mathcal{F}_1, \ldots, \mathcal{F}_m$ are closely related. For example, consider the following case,

```
P1’   select orderID from order where custID=#$CUSTID;
P2’   select orderID from order where odate=#$ODATE;
P3’   select orderID from order where itemid=#$ITEMID;
```

These three fingerprints can derive the following:

```
P4’   select orderID from order where custID=#$CUSTID and odate=#$ODATE and itemid=#$ITEMID;
```

of compactness 1. The three fingerprints (P1’,P2’,P3’) are likely generated by different functions. If this is the case, the derived fingerprint (P4’) will never appear in the trace log.

To search for derived fingerprints with a high degree of compactness, we perform the following steps,

1. Group all the legitimate fingerprints into equivalent classes such that each pair of fingerprint F_i, F_j in the same class differs only by some missing conjuncts in their WHERE-clauses.
2. For each equivalent class, conjunct all the conditions in the WHERE-clauses of all fingerprints, removing duplicate conjuncts if necessary, into a condition C. Construct F_0 such that it is identical as F_1 except that the WHERE-clause is replaced with C.

Note that the legitimacy of this derived fingerprint should be confirmed with the DBA, before including it in the set of legitimate fingerprints.

5 Algorithm for Fingerprint Learning and Derivation

In this section, we summarize the discussion in the previous sections by presenting the algorithm to automatically generate the set of fingerprints. We illustrate the algorithm with a complete example.

The algorithm is shown in Algorithm 1. After that is a detailed walkthrough of the algorithm using the trace log shown in Figure 2.

Algorithm 1: Algorithm for Fingerprint Learning and Derivation

Given trace log *LOG*, and a set of attributes $\mathcal{A}$ that carries some implicit meaning optionally specified by the DBA, we perform the following steps:

1. For each attribute in the WHERE-clauses of *LOG* that is not found in $\mathcal{A}$, scan *LOG* to compute the Kolmogorov-Smirnov's D statistic to test for linear growth of the number of unique literals.
2. If the number of unique literals of any attribute does not statistically support a linear growth, put it into the set $\mathcal{A}'$.
3. Generate a set of unique fingerprints by replacing all the literals of attributes in the WHERE-clause that are neither in $\mathcal{A}$ nor $\mathcal{A}'$, with meta-constants.
   (a) replace all the string literals of the expression `attr='literal'` in the WHERE-clause with `attr=#$attr`.
   (b) replace all the numeric literals of the expression `attr=numeric_literal` in the WHERE-clause with `attr=#%attr`.
   Let $\mathcal{F}_1, \ldots, \mathcal{F}_m$ be the fingerprints generated.
4. Count the number of occurrences for each fingerprint. Label the fingerprint "safe" if the number of occurrences is not statistically 0.
5. For each unclassified fingerprint $\mathcal{F}$, label it "safe" if there exists another "safe" fingerprint $\mathcal{F}'$, such that $\mathcal{F}$ differs from $\mathcal{F}'$ by only
   (a) any extra condition in the WHERE-clause of $\mathcal{F}$ that is missing from $\mathcal{F}'$ is joined with the "AND" operator; and
   (b) $\mathcal{F}$ selects an equal number of or fewer columns than $\mathcal{F}'$.
   The remaining fingerprints are labelled "unsafe".
6. Compute the set of derived fingerprints from the set of "safe" fingerprints. Mark these derived fingerprints as "derived".
7. Present all the "unsafe" and "derived" fingerprints to the DBA. Each "unsafe" or "derived" fingerprint that is approved by the DBA will be marked as "safe". If there remains any "unsafe" fingerprints, then an intrusion has occurred in *LOG*.
8. For each "safe" fingerprint, replace the meta-constant by the regular expressions. This legitimate fingerprint database can now be used to detect intrusions for the database application.

1. Assuming $\mathcal{A}$ is an empty set, we first scan the log to compute the Kolmogorov-Smirnov's D statistic for discrete data against linear growth

Table 2. Occurrence count of the fingerprints

Fingerprint	F1	F2	F3	F4	F5	F6	F7	F8
Frequency	5	1	3	1	4	1	1	1

of the number of unique values for each attribute in the WHERE-clause (i.e. `custid`, `amt`, `prodid`, `conf`, `odate`). For example, (as calculated in the previous section) the `prodid` has a D statistic of 0.15, while the `conf` has a D statistic of 0.6.

2. After step 2, we have $\mathcal{A}' = \{\texttt{conf}\}$.
3. The next step is to generate the initial fingerprint set by replacing the literals of each attribute not in $\mathcal{A}$ nor $\mathcal{A}'$ with its corresponding meta-constant. We obtain the following fingerprints:

```
F1   select orderID, Amt from order where custid=#$CUSTID;
F2   select orderID from order where custid=#$CUSTID and amt>#%AMT and amt<#%AMT;
F3   select prod_desc from product where prodid=#$PRODID;
F4   select orderID from order where custid=#$CUSTID or TRUE or custid=#$CUSTID;
F5   select comment from prod_comment where prodid=#$PRODID and conf='PUBLIC';
F6   select comment from prod_comment where prodid=#$PRODID and conf='PRIVATE';
F7   select orderID from order where custid=#$CUSTID and odate=#$ODATE and amt>#%AMT;
F8   select orderID from order where custid=#$CUSTID and odate=#$ODATE and amt<#%AMT;
```

4. We count the number of occurrences for each fingerprint next. The results are tabulated in Table 2.

   F1, F3 and F5 occurs frequently enough for us to infer that they are "safe". On the other hand, F2, F4, F6, F7 and F8, occurs only once each in the trace log. These fingerprints need to be looked into.
5. Upon inspection, F2, F7 and F8 can be classified as "safe" as they can be subsumed by F1. F4 and F6, however, are marked "unsafe". They should be highlighted to the DBA.
6. The next step involves finding derived fingerprints of high compactness. In this example, we set the degree of compactness to be at least 2. The following new fingerprint marked "derived" is found,

```
F9   select orderID from order where custid=#$CUSTID and odate=#$ODATE
        and amt>#%AMT and amt<#%AMT;
```

7. Assume the DBA approves the "derived" fingerprint F9, but does not approve all the other "unsafe" fingerprints. The legitimate fingerprints are F1, F2, F3, F5, F7, F8 and F9. The meta-constants are replaced by the appropriate regular expressions. The final fingerprints set of legitimate fingerprints for this database application consists of:

```
F1   select orderID, Amt from order where custid='[^']*';
F2   select orderID from order where custid='[^']*' and amt>[0-9]+ and amt<[0-9]+;
F3   select prod_desc from product where prodid='[^']*';
F5   select comment from prod_comment where prodid='[^']*' and conf='PUBLIC';
F7   select orderID from order where custid='[^']*' and odate='[^']*' and amt>[0-9]+;
F8   select orderID from order where custid='[^']*' and odate='[^']*' and amt<[0-9]+;
F9   select orderID from order where custid='[^']*' and odate='[^']*' and amt>[0-9]+
        and amt<[0-9]+;
```

6 Related Work

As far as the authors know, this is the only work using SQL transaction fingerprints or signatures to detect database intrusions. Closest to our work is [15] which profiles users and roles in a relational database system. It generates the profiles from the “working scopes” of the users which are defined as the sets of attributes that are usually referenced together with some values. This profile describes typical user behaviour and is used to detect misuse. This method assumes that the legitimate users show some level of consistency in using the database system. If this assumption does not hold, or if the threshold for inconsistency is not set properly, the result will be a high level of false positives. This method also faces the attribute selection problem when it chooses the “features” to consider when building the working scopes.

Classification algorithms (e.g. C4.5 [16]) seem to be applicable to our problem of identifying the instances of SQL statements that constitute an intrusion. However, this will lead to the “feature selection” problem in which we have to decide on the feature set of the SQL statements to be used in the classifier. The entire training set will also have to be tagged as either legitimate or illegitimate which may be infeasible for large training sets. Text summarization techniques [17, 18] cannot be applied to our problem of SQL transaction summarization as unlike text collections, our logs consist of largely independent SQL transactions and lack the dependency (style, theme) among linguistic units (phrase, clause, etc) in text collections.

7 Conclusion

DIDAFIT is a database intrusion detection system that identifies anomalous database accesses by matching database transactions with a set of legitimate transaction fingerprints. This work addresses the problem of learning the set of legitimate fingerprints from the database trace logs that contain the SQL statements. We developed an algorithm that can:

1. selectively and effectively summarize SQL statements into fingerprints.
2. detect “high-risk” (possibly malicious) SQL statements even in the training set of SQL statements.
3. derive possibly legitimate fingerprints that are missing from the SQL statements in the training set.

Currently, the algorithm works mostly at the syntactic-level. We are looking into how domain knowledge can contribute to form a semantically-richer intrusion detection mechanism. Further research is also done to study how the algorithm can be extended to support incremental learning.

Acknowledgment

The authors would like to thank the anonymous reviewers for their useful comments.

References

[1] Low, W. L., Lee, S. Y., Teoh, P.: DIDAFIT: Detecting Intrusions in Databases Through Fingerprinting Transactions. In: Proceedings of the 4th International Conference on Enterprise Information Systems (ICEIS). (2002) 264, 265, 267, 269

[2] Atanasov, M.: The truth about internet fraud. In: Ziff Davis Smart Business, Available at URL `http://techupdate.zdnet.com/techupdate/stories/main/0,14179,2688776-11,00.html` (2001) 264

[3] Hatcher, T.: Survey: Costs of computer security breaches soar. In: CNN.com, Available at URL `http://www.cnn.com/2001/TECH/internet/03/12/ csi.fbi.hacking.report/` (2001) 264

[4] Poulsen, K.: Guesswork Plagues Web Hole Reporting. In: SecurityFocus, Available at URL `http://online.securityfocus.com/news/346` (2002) 264, 265

[5] Internet Security Systems: RealSecure Intrusion Detection Solution, Available at URL `http://www.iss.net` (2001) 265

[6] NFR Security: NFR network intrusion detection, Available at URL `http://www.nfr.com/products/NID/` (2001) 265

[7] Enterasys Networks, Inc.: The Dragon IDS, Available at URL `http://www.enterasys.com/ids/dragonids.html` (2001) 265

[8] Cisco Systems, Inc.: Cisco Intrusion Detection, Available at URL `http://www.cisco.com/warp/public/cc/pd/sqsw/sqidsz/` (2001) 265

[9] Symantec Corporation: Enterprise Solutions, Available at URL `http://enterprisesecurity.symantec.com/` (2001) 265

[10] Roesch, M.: Snort: Lighweight intrusion detection for networks. In: Proceedings of the 13th Conference on Systems Administration (LISA-99), USENIX Association (1999) 229–238 265

[11] Andrews, C.: SQL injection FAQ, Available at URL `http://www.sqlsecurity.com` (2001) 266

[12] Anley, C.: Advanced SQL Injection In SQL Server Applications, Next Generation Security Software Ltd, Available at URL `http://www.nextgenss.com/papers/advanced_sql_injection.pdf` (2002) 266

[13] Anley, C.: (more) Advanced SQL Injection, Next Generation Security Software Ltd, Available at URL `http://www.nextgenss.com/papers/more_advanced_sql_injection.pdf` (2002) 266

[14] Oracle: Oracle, 2001, Available at URL `http://www.oracle.com` (2001) 269

[15] Chung, C. Y., Gertz, M., Levitt, K.: Misuse detection in database systems through user profiling. In: Web Proceedings of the 2nd International Workshop on the Recent Advances in Intrusion Detection (RAID). (1999) 278

[16] Quinlan, J. R.: Induction of decision trees. In Shavlik, J. W., Dietterich, T. G., eds.: Readings in Machine Learning. Morgan Kaufmann (1990) Originally published in *Machine Learning* 1:81–106, 1986. 278

[17] Hovy, E., Lin, C. Y.: Automated Text Summarization in SUMMARIST. In: Proceedings of ACL/EACL Workshop on Intelligent Scalable Text Summarization. (1997) Madrid, Spain. 278

[18] Boguraev, B., Bellamy, R.: Dynamic Presentation of Phrasally-Based Document Abstractions. In: Proceedings of Thirty-second Annual Hawaii International Conference on System Sciences (HICSS). (1998) 278