Abstract
Comparing where sub-clauses between different SQL statements is important in analyzing SQL statements, which is usually the key for databases security. Comparing them literally is not sufficient, because there may be different character strings for the same semantic structure. A method called Logical Tree (LT) is proposed in this paper, which is used to compare different where sub-clauses semantically. By this method, where sub-clauses of SQL statements are denoted by trees. Some basic definitions of LT are addressed. According to laws of logical expressions, which are the form of where sub-clauses, accordingly laws and operations are defined for LT. When different LTs are reduced basing on some principles, they can be compared according to their structures. As a result, the where sub-clauses they denote are compared semantically. Basing on the method of LT, there may be more work about analyzing SQL statements can be done.

Keywords: SQL parsing, Database security, Intrusion detection, Database misuse

1. Introduction

Analyzing and comparing SQL statements is important in detecting malicious database operations. Attackers always design special SQL statements with special where sub-clauses to achieve their illegal purposes. Mining legal users’ profiles in SQL statement logs, and then comparing new SQL statements to the profiles, is usually used by database security administrators.

When comparing different SQL statements, the common method is abstracting their patterns literally, which is called signature or fingerprint of SQL statements. Early in 2002, the authors of [1] and [2] had defined the signature of a SQL statement as SQL fingerprint for detecting SQL injection attack, which is usually be used in database based application like that of [3]. SQL statements can be compared with the fingerprints, so that it can be known whether the SQL statements are malicious. In [4], the signature of a SQL statement is calculated by a hash algorithm. In [5], skeleton query is generated by replacing all occurrences of constants in the query with an empty place holder token, but not changing. M. Vieira and H. Madeira used logs of DBMS to get users’ transaction profile, and then find malicious transactions [6]. K. Kemalitis and T. Tzouramanis [7] developed a specification-based approach for SQL injection detection for web applications. They define some structures of specific SQL statements, and then compare the input statements to them. In fact, the main ideas of researching on detecting SQL injection attacks can be similarly summarized as removing the values of a SQL query, and then comparing it with a predefined ones [8][9][10], or machine learning and classification [11]. The authors of [12] and [13] applied parsing tree on SQL statements to tackle SQL injection attacks, but basically literally only. The authors of [14] and [15] implement their method in programming. [14] describes a more natural style of programming that yields code that is impervious to injections by construction. The authors of [15] developed a tool between the application and its underlying relational DBMS. That tool uses stripped-down SQL queries and stack traces to create SQL statement signatures that are then used to distinguish between injected and legitimate queries. Recently, A. Tajpour, S. Ibrahim, and M. Masrom present SQL injection attack types and techniques for them. They also evaluate the techniques in [16]. In our previous work about DDM [17][18], we also gave some definitions for abstracting patterns of SQL statements. In our work, by replacing the specific data values in a SQL statement with a uniform token $T$, the pattern of the statement can be obtained, which is denoted by $P$. After that, the patterns of two SQL statements can be compared by separately comparing their operation, their set of relevant relations, their set of relevant attributes, and their where sub-clauses with specific data values in them replaced by a uniform token. By comparing,
it can be known whether the two SQL statements are of the same pattern, and then the algorithm of DDM will continue working. During that process, comparing different where sub-clauses is a key step both in mining profiles and checking new SQL statements. However, as we can see from the above, comparing SQL statements patterns literally is usually used for tackling SQL injection attacks. Even in our previous work, we cannot compare different patterns semantically, because the where sub-clause of a SQL statement is always a logical expression. That means, if we compare two where sub-clauses simply by comparing character strings or regular expression as [1] and [2] do, we may conclude that they are different even though they are the same semantically.

For example, there are two SQL statements, which are called $S_1$ and $S_2$ as follows:

\[
S_1=\text{Select col1, col2 from table1 where col3=5 and col4='Bill'};
\]
\[
S_2=\text{Select col1, col2 from table1 where col4='Gates' and col3=10};
\]

By replacing the specific values in $S_1$ and $S_2$ with a uniform token $STS$, $P_1$ and $P_2$, which are the patterns of $S_1$ and $S_2$ respectively, are generated as follows:

\[
P_1=\text{Select col1, col2 from table1 where col3=$STS$ and col4=$STS$};
\]
\[
P_2=\text{Select col1, col2 from table1 where col4=$STS$ and col3=$STS$};
\]

Obviously, if we compare $P_1$ and $P_2$ simply by comparing character strings or regular expression, we will conclude that they are different, but they are the same semantically in fact. If we call the part of where sub-clause in a pattern as $sub-P$, comparing $sub-P$s semantically is important to database security by discovering malicious SQL statements. Fortunately, $sub-P$, which is composed of several predicate logic expressions connected by logical operators such as and, or, not, in, etc. Basing on that character of $sub-P$, we propose a method to compare $sub-P$s semantically by tree. The idea of parsing tree can be applied here, because it makes us to analyze and compare SQL statements semantically in their structures. However, previous work about parsing tree only parses SQL statements statically. We think parsing tree should be used more flexible, because a logical expression can be rewrote in several forms literally. To start from focusing on where sub-clauses of SQL statements in a distinguishing way from parsing tree, we propose a methodology which is called as Logical Tree. In this way, $sub-P$s are not compared literally, but are compared in their structure, namely in semantic.

2. Logical Tree

Different predicate logic expressions may be the same semantically, so they should be compared semantically, but not literally. For example, $(A \land B) \land (x > 10)$ is different from $A \land ((10 < x) \land B)$ in word, but they have the same semantic and result. Hence, Logical Tree is proposed to record predicate logic expression and compare them. The following are basic definitions.

**Definition 1 (Meta-Item).** In the context of standard SQL, $Exp_1$ and $Exp_2$ denote two expressions, while $opr$ denotes a comparing operator. A Meta-Item is defined as an expression in the form of $Exp_1 \ opr \ Exp_2$. $Exp_1$ and $Exp_2$ may be an attribute of a relation in the database, variable, value, or algebraic expression, while $opr$ may be $>$, $<$, $\leq$, $\geq$, $=$ or like. For example, $x > 10$, $a$ like ‘hello%’ are both Meta-Items.

It should be noticed that $opr$ cannot be $\neq$ and $\equiv$, or other symbol, because they cannot be used in standard SQL. The symbol $\neq$ can be denoted by logic NOT of symbol $\equiv$. For example, $b \neq 5$ is denoted by $\neg (b = 5)$. In the case of the operator $\neg$ acting on a single Meta-Item, the operator is considered as a part of the Meta-Item. On the other hand, when operator $\neg$ acts on an expression in a Meta-Item, it is considered as a part of the expression on which it acts.

**Definition 2 (MIT).** Given a Meta-Item, we take its $opr$ as the root of a binary tree, while its $Exp_1$ and $Exp_2$ are left sub-tree and right sub-tree of the root. This binary tree is named as Meta-Item Tree (MIT). If operator $\neg$ acts on a single Meta-Item, a node labeled by $\neg$ is added above the root of MIT to be the new root. If operator $\neg$ acts on an expression in a Meta-Item, a node labeled by $\neg$ is added as the root of the sub-tree for which the expression stands.

**Example 1.** The MITs of $\neg (b = 5)$ and $\neg b = 5$ are illustrated in Figure 1 as follows:
Additionally, a single expression without any comparing operator will generate a special type of MIT, which has only one node labeled by the expression. According to the properties of the comparing operators, some reversing laws of MIT are defined as follows.

**Law 1.** Because of \( x > y \Leftrightarrow y < x \), it is defined:

\[
\begin{array}{c}
> \\
\hline
x & y
\end{array}
\Leftrightarrow
\begin{array}{c}
< \\
\hline
y & x
\end{array}
\]

**Law 2.** Because of \( x \geq y \Leftrightarrow y \leq x \), it is defined:

\[
\begin{array}{c}
\geq \\
\hline
x & y
\end{array}
\Leftrightarrow
\begin{array}{c}
\leq \\
\hline
y & x
\end{array}
\]

**Law 3.** Because of \( x = y \Leftrightarrow y = x \), it is defined:

\[
\begin{array}{c}
= \\
\hline
x & y
\end{array}
\Leftrightarrow
\begin{array}{c}
= \\
\hline
y & x
\end{array}
\]

In the context of standard SQL, operator *like* does not satisfy reversing laws.

**Definition 3 (CMI and DMI).** An expression composed by two Meta-Items connected by a conjunction operator \( \land \), is named as Conjuntion Meta-Item (CMI). On the other hand, an expression composed by two Meta-Items connected by a disjunction operator \( \lor \), is named as Disjuntion Meta-Item (DMI). For example, \( \neg(b = 5) \lor (a \leq 10) \) is a CMI, while \( (b \geq 5) \land (b \leq 11) \) is a DMI.

**Definition 4 (CMIT and DMIT).** Given a CMI, we take its conjunction operator \( \land \) as the root of a binary tree, while its left and right MIT separately labeled by its left and right MI are left sub-tree and right sub-tree of the root. This binary tree is named as Conjunction Meta-Item Tree (CMIT). Correspondingly, if the conjunction operator \( \land \) is replaced by a disjunction operator \( \lor \), the generated binary tree is named as Disjuntion Meta-Item Tree (DMI). If operator \( \neg \) acts on a CMIT or a DMIT, a node labeled by \( \neg \) is added above the root of CMIT or DMIT to be the new root.

Given P and Q as two MITs, we have following examples:

**Example 2.** The CMIT of \( P \land Q \) and the DMIT of \( \neg(P \lor Q) \) are illustrated in Figure 2 as follows:

\[
\begin{array}{c}
\land \\
\hline
P & Q
\end{array}
\]

(a) CMIT of \( P \land Q \)

\[
\begin{array}{c}
\lor \\
\hline
P & Q
\end{array}
\]

(b) DMIT of \( \neg(P \lor Q) \)

**Figure 2.** Examples of CMIT and DMIT
Definition 5 (LT). A Logical Tree (LT) is a tree, whose root is labeled by a conjunction operator $\land$ or a disjunction operator $\lor$, with its sub-trees being MIT, CMIT, DMIT, or LT. By this definition, a CMIT or a DMIT is the smallest LT.

Given $P$ and $Q$ as MIT, CMIT, DMIT, or LT, basing on the commutative law, associative law, distributive law of conjunction and disjunction, and De Morgan's laws, corresponding operation on LT are defined as follows:

Operation 1 (commutative law of conjunction). Because of $P \land Q \Leftrightarrow Q \land P$:

$$
\begin{array}{c}
\land \\
P \\
Q \\
\land \\
\end{array}
\quad
\begin{array}{c}
\land \\
Q \\
P \\
\land \\
\end{array}
$$

Operation 2 (commutative law of disjunction). Because of $P \lor Q \Leftrightarrow Q \lor P$:

$$
\begin{array}{c}
\lor \\
P \\
Q \\
\lor \\
\end{array}
\quad
\begin{array}{c}
\lor \\
Q \\
P \\
\lor \\
\end{array}
$$

Operation 3 (associative law of conjunction). Because of $(P \land Q) \land R \Leftrightarrow P \land (Q \land R)$:

$$
\begin{array}{c}
\land \\
\land \\
P \\
Q \\
R \\
\land \\
\land \\
\end{array}
\quad
\begin{array}{c}
\land \\
\land \\
Q \\
P \\
\land \\
R \\
\land \\
\land \\
\end{array}
$$

Operation 4 (associative law of disjunction). Because of $(P \lor Q) \lor R \Leftrightarrow P \lor (Q \lor R)$:

$$
\begin{array}{c}
\lor \\
\lor \\
P \\
Q \\
R \\
\lor \\
\lor \\
\end{array}
\quad
\begin{array}{c}
\lor \\
\lor \\
P \\
P \\
Q \\
R \\
\lor \\
\lor \\
\end{array}
$$

Operation 5 (distributive law). Because of $P \lor (Q \land R) \Leftrightarrow (P \lor Q) \land (P \lor R)$:

$$
\begin{array}{c}
\lor \\
P \\
\land \\
Q \\
R \\
\lor \\
\land \\
\end{array}
\quad
\begin{array}{c}
\lor \\
P \\
\land \\
Q \\
P \\
\land \\
R \\
\lor \\
\land \\
\end{array}
$$

And because of $P \land (Q \lor R) \Leftrightarrow (P \land Q) \lor (P \land R)$:

$$
\begin{array}{c}
\land \\
P \\
\lor \\
Q \\
R \\
\land \\
\land \\
\end{array}
\quad
\begin{array}{c}
\land \\
P \\
\lor \\
Q \\
P \\
\land \\
R \\
\land \\
\land \\
\end{array}
$$

Operation 6 (De Morgan's laws). Because of $\neg(P \lor Q) \Leftrightarrow \neg P \land \neg Q$:
And because of \( \neg(P \land Q) \equiv \neg P \lor \neg Q \):

Furthermore, basing on the associative law of conjunction and disjunction, reduction operation is defined for LT as follows.

**Operation 7 (Merging).** In an LT, if two nodes whose relationship is parent and child, and they are both labeled by conjunction operator \( \land \) or disjunction operator \( \lor \), these two nodes can be merged into one node. The new LT generated is equal to the old one. This operation is illustrated as follows:

By this operation, the node labeled by conjunction operator \( \land \) or disjunction operator \( \lor \) will have more than two sub-trees. That does not mean conjunction operator \( \land \) and disjunction operator \( \lor \) becomes multivariate operators. It’s just shortened form of continuous conjunction or disjunction. The multiple sub-trees of the node labeled by conjunction operator \( \land \) or disjunction operator \( \lor \) satisfies commutative law and associative law of conjunction and disjunction.

### 3. Comparing LTs

As mentioned previously, sub-\( P \)s can be compared in semantic by comparing their LTs. In this section, the way of comparing two LTs is stared.

#### 3.1. Steps of comparing LTs

Before being compared, two LTs must be reduced, which is an important step. The purpose of reducing an LT is to reduce the number of nodes, to simplify the structure of the LT, and to unify the form of LT. Those three purposes result in following principles:
Principle 1: To replace the sub-trees which satisfy associative law and De Morgan’s laws of LT with equivalent ones. The principle of replacing is reducing the number of nodes.

Principle 2: By taking the operation of Merging, to merge the two nodes two nodes whose relationship is parent and child, and both labeled by conjunction operator $\land$ or disjunction operator $\lor$. By this principle, the number of nodes and levels of the LT can be reduced, and the structure of the LT is also simplified.

Principle 3: To reduce nodes labeled by comparing operators by reversing laws of MIT, so that the number of types of nodes labeled by comparing operators is reduced. For example, by reversing all MITs with root of $\geq$, those MITs are all replaced by those with root of $\leq$.

Principle 4: To sort the sub-trees of nodes labeled by conjunction operator and disjunction operator, basing on Operation 1 and Operation 2.

When sorting the sub-trees in Principle 4, their priority of being left sub-trees is defined according to the labels of their roots, as follows: $=$, $<$, $\leq$, $\geq$, $>$, like, $\neg$, $\land$, $\lor$. If the labels of two roots are the same, sort them on the labels of their sub-trees recursively.

By the four principles above, the differences on structure between different LTs are reduced. After that, two LTs can be compared by BFS (Bread First Searching). The reason of choosing BFS is to discover the differences between different LTs semantically as soon as possible.

### 3.2. Example of comparing LTs

An example illustrates the procedure of comparing two sub-Ps by LT as follows.

**Example 3.** The are two sub-Ps:

- $sp_1 = \text{col1} \lt \text{T} \text{S} \text{S} \land (\text{col2} \leq \text{T} \text{S} \text{S} \lor \text{col2} \geq \text{T} \text{S} \text{S})$ and $\text{col1} \geq \text{T} \text{S} \text{S}$
- $sp_2 = \text{T} \text{S} \text{S} \leq \text{col1} \land ((\text{col1} \lt \text{T} \text{S} \text{S} \land \text{col2} \leq \text{T} \text{S} \text{S}) \lor (\text{col1} \lt \text{T} \text{S} \text{S} \land \text{col2} \geq \text{T} \text{S} \text{S}))$

Their LTs are:

\[
\text{LT}_1 = \begin{array}{c}
\land \\
\langle \\
\langle \\
\text{col1} & \text{T} \text{S} \text{S} \\
\leq \\
\text{col2} & \text{T} \text{S} \text{S} \\
\lceil \\
\text{col1} & \text{T} \text{S} \text{S} \\
\end{array}
\]

and

\[
\text{LT}_2 = \begin{array}{c}
\land \\
\leq \\
\langle \\
\langle \\
\text{col1} & \text{T} \text{S} \text{S} \\
\text{col2} & \text{T} \text{S} \text{S} \\
\rangle \\
\text{col1} & \text{T} \text{S} \text{S} \\
\end{array}
\]

$LT_i$ is reduced as follows:

**Step 1:** By merging $LT_i$ changes into:
Step 2: By reversing laws of MIT $LT_i$ changes into:

Step 3: By commutative law, the sub-trees are sorted, and $LT_i$ changes into:

The reducing of $LT_1$ is completed.

$LT_2$ is reduced as follows:

Step 1: By associative law $LT_2$ changes into:

Step 2: By merging $LT_2$ changes into:
Step 3: By reversing laws of MIT $LT_2$ changes into:

$LT_2 = \begin{array}{c}
\land \\
\leq \\
\lt \\
ST \texttt{ col1} \\
\leq \\
\gt \\
ST \texttt{ col2} \\
\end{array}$

Step 4: By commutative law, the sub-trees are sorted, and $LT_2$ changes into:

$LT_2 = \begin{array}{c}
\land \\
\leq \\
\lt \\
ST \texttt{ col1} \\
\leq \\
\lt \\
ST \texttt{ col2} \\
\end{array}$

The reducing of $LT_2$ is completed.

By utilizing BFS, it can be easily seen that the two reduced LTs are the same semantically.

4. Conclusion

When comparing two SQL statements, the comparison of their where sub-clauses is very important. By replacing the specific values in where sub-clauses with a uniform token $ST$, patterns of where sub-clauses are generated. A method of comparing patterns of different where sub-clauses is proposed by this work, which is called Logical Tree, and shortened as LT. By that method, different patterns of where sub-clauses can be compared semantically, but not only literally.

In the future, there can be at least three more work to do about LT: (1) The method of LT can be developed more to fit more complex where sub-clauses. (2) New method together with LT can be developed to compare whole SQL statements. (3) The dissimilarity of different SQL statements should be measured quantitatively by the method about LT. In addition, when the method of LT is completed in theory, it should be practiced in real database operations.

Acknowledgments

This work is supported by the Fundamental Research Funds for the Central Universities grants ZZ1224, China.
References


