Journal of Network and Systems Management, Vol. 13, No. 3, September 2005 (© 2005)
DOI: 10.1007/s10922-005-6264-1

Design and Analysis of Techniques for Detection of Malicious Activities in Database Systems

Yi Hu[1] and Brajendra Panda[1,2]

Existing host-based Intrusion Detection Systems use the operating system log or the application log to detect misuse or anomaly activities. These methods are not sufficient for detecting intrusion in the database systems. In this paper, we describe a method for detecting malicious activities in a database management system by using data dependency relationships. Typically, before a data item is updated in the database, some other data items are read or written. And after the update, other data items may also be written. These data items read or written in the course of update of a data item construct the read set, prewrite set, and the postwrite set for this data item. The proposed method identifies malicious transactions by comparing these sets with data items read or written in user transactions. We have provided mechanisms for finding data dependency relationships among transactions and use Petri-Nets to model normal data update patterns at user task level. Using this method, we ascertain more hidden anomalies in the database log. Our simulation on synthetic data reveals that the proposed model can achieve desirable performance when both transaction and user task level intrusion detection methods are employed.

KEY WORDS: Malicious transactions; intrusion detection; anomaly detection; data dependency.

1. INTRODUCTION

A secure information system has three important features: prevention, detection, and recovery [1]. Intrusion Detection is employed to detect the malicious activities in case the system prevention mechanism fails. Almost all current host-based anomaly detection approaches are based on the data generated by auditing mechanism of the operating system or application. But these data do not sufficiently reflect what special data items in the system are modified, e.g., what particular

[1]Computer Science and Computer Engineering Department, University of Arkansas, Fayetteville, Arkansas 72701.
[2]To whom correspondence should be addressed at Computer Science and Computer Engineering Department, University of Arkansas, Fayetteville, Arkansas 72701. E-mail: bpanda@uark.edu

1064-7570/05/0900-0269/0 © 2005 Springer Science+Business Media, Inc.

attributes in the database are read or written, and whether it is legal to modify these data items at that time.

Our database intrusion detection system tries to find malicious database transactions submitted to the Database Management System (DBMS) by an intruder masqueraded as a normal user. The malicious transactions identified in this work can be used later by damage evaluation and recovery procedures [2–4]. In our approach, we concentrate on analyzing the *dependencies* among the data items in the database. By data dependency, we refer to the data access correlations between two or more data items. That is, which data items must be read or written before a data item gets updated and which others are written after the update. Two semantic analyzers are proposed in this work to deduce the data dependencies among data items accessed by transactions in the database application program. By checking whether each update operation in the user transaction conforms to the data dependencies, generalized anomalous activities at the transaction level are detected. We also propose a method for finding anomalies at the user task level by using Petri-Nets to model normal data update sequences in user tasks. This is especially useful for finding hidden malicious activities that consist of several transactions, each of which appears as a normal transaction.

The following section describes past efforts related to our research. Section 3 outlines the model. Static and dynamic semantic analyzers are discussed in Section 4. The notion of data dependency at user task level is described in Section 5 and an implementation approach using Petri-Nets is presented in Section 6. Section 7 provides the simulation model and performance results. Section 8 presents the conclusions.

2. RELATED WORK

Currently, most research on intrusion detection concentrates on anomaly detection in computer systems. The method used by IDES [5, 6] tries to construct user profiles statistically based on different categories of system parameters. These different parameters are weighted and combined together to generate a vector to reflect normal user behavior. By comparing it with current user profile created from history data, anomaly activities could be found. Other techniques [7, 8] attempt to profile normal program behavior. They try to find the normal system call sequences and save these sequences as normal program access patterns in the database. Later these patterns are used to check with program activities to find any anomalous activities. The method developed by Lane and Brodley [9] creates normal user command sequence. It's based on the observation that users generally make use of fixed set of UNIX commands, and different users have different ways of using these commands.

Since generating normal user or program profiles is a gradual training process, learning the profile through the process can be considered as a machine learning

procedure. So, artificial intelligence application in intrusion detection is employed by some researchers [10]. Data mining [11–13] and neural network are used to make the IDS more intelligent.

Very limited research has been conducted in the field of database intrusion detection. Lee *et al.* [14] have used time signatures in discovering database intrusions. Their approach is to tag the time signature to data items. A security alarm is raised when a transaction attempts to write a temporal data object that has already been updated within a certain period. Another method presented by Chung *et al.* [15] identifies data items frequently referenced together and saves this information for later comparison. If the system observes substantial incidents of data items that are referenced together, but not in normal patterns as established before, an anomaly is signalled.

3. THE MODEL

The proposed model is designed to identify malicious transactions submitted to the DBMS by an intruder masquerading as a normal user. In various different ways, an intruder can gain access to a database system, such as obtaining the password of a normal user. In such cases, the database system cannot distinguish between an intruder and a normal user. Here normal user activity means a user accessing data items through a DBMS by using database application program instead of by submitting transactions to the DBMS manually and these activities do not include system maintenance and administration activities. For example, the accountants in a bank use the banking application to do daily customer transactions such as deposit, withdrawal, account transfer, and loan. By entering information to the program interface of the banking application, e.g. GUI of the application program, the accountant can access and update account information of customers.

3.1. Database Model

This work is based on the relational database model [16]. A transaction is a logical unit of database processing that includes one or more database access operations. We only consider the transactions that do not contain conditional statements, i.e., *if. . .then. . .else* statements, to simplify the process of data dependency analysis. For the transaction that has conditional statements, multiple subtransactions each of which only containing one sequential execution path are generated and used for data dependency analysis. Our model requires that the database log not only records the database access operations in each transaction, but also keeps the identification information of the user submitting the transaction to the DBMS. We also require that timestamps associated with each transaction indicating the transaction start and end times are also kept in the database log.

The database log records not only the write operations but also the read operations of each transaction. In case some log information hasn't been stored on the permanent storage devices before a transaction commits, our database intrusion detection system can still access the contents of the log for uncommitted transactions from the temporary log.

The user's task in this research refers to a group of transactions that are always submitted to the DBMS together to achieve a certain goal. For example, in order to perform the account transfer in a banking application, several transactions may be submitted to the database consecutively to fulfill the task.

3.2. Assumptions

We assume that the intruder has no access to the database application program that a normal user has. So an intruder cannot submit malicious transactions to the DBMS through the database application program at the normal user site. This is the case when an intruder accesses the database from a remote site by submitting transactions manually or through a different application. For example, the intruder may have obtained the password of a legitimate database user account.

The total number of transactions a normal user can use is limited. They are the transactions in the database application program. So in this case our database intrusion detection system will raise alarm when the database administrator does some legal modification to the database system that doesn't conform to the data dependencies observed. In this case, database administrator is responsible for identifying whether the cause of the alarm is due to a normal database maintenance work or due to the intruder masquerading as the database administrator performing malicious activities.

4. SEMANTIC ANALYZERS

The static and dynamic semantic analyzers perform the analysis of the data dependencies among data items in the database. The following definitions help in understanding the concept.

Definition 1. The *Read Set* for one data item is a set of different data item sets, each of which consists of zero or more ordered data items. Additionally, the transaction must read all data items in one data item set of the read set *before* the transaction updates this data item. The notation $rs(x)$ is used to denote the read set for data item x. The read set is used to calculate the new value of this data item for updating purposes. For example, consider the following update statement.

$$Update\ Table1\ set\ x = a + b + c\ where \ldots$$

In this statement, before updating x, the values of a, b, and c must be read and added together to get the new value of x. So $\{a, b, c\} \in rs(x)$.

Since when constructing the read set for x we only consider the data items read and used for the purpose of calculating the new value of x, we don't need to consider the data items read in the *where* statements. That's why the part in the *where* statement is ignored.

Definition 2. The *Prewrite Set* for one data item is the set of different data item sets, each of which consists of zero or more ordered data items. Furthermore, the transaction must write all data items in the specified order in one data item set of the prewrite set *before* the transaction updates this data item. The notation $ws0(x)$ is used to denote the prewrite set of data item x. The reason we define prewrite set is that sometimes a data item is updated in the database after other data items are updated. For example, we have three update statements in one transaction in the following sequence. Note that one SQL statement doesn't necessarily immediately follow the other; there can be other nonupdating SQL statements between them.

$$Update\ Table1\ set\ x = a + b + c\ where \ldots$$
$$Update\ Table1\ set\ y = x + u\ where \ldots$$
$$Update\ Table1\ set\ z = x + w + v\ where \ldots$$

It must be noted that when x is updated, y and z are updated subsequently. Because of the hard-coded sequence, the transaction always updates y before updating z. So considering data item z, we observe that $\{x, y\}$ belongs to its prewrite set, that is $\{x, y\} \in ws0(z)$.

Definition 3. The *Postwrite Set* for one data item is the set of different data item sets, each of which consists of zero or more ordered data items. Moreover, the transaction must write all data items in the specified order in one data item set of the postwrite set *after* the transaction updates this data item. Using the above example, it can be noted that $\{y, z\}$ belongs to the postwrite set of data item x, that is $\{y, z\} \in ws1(x)$, where *ws1* denote the postwrite set.

4.1. Static Semantic Analyzer

The static semantic analyzer is used to analyze the database application program statically to decide the read set, the prewrite set, and the postwrite set. Here the word static is used to describe that our method is based on the transaction program, not based on the database log. First, we want to find out all the possible transactions one user may use. These transactions can be identified by checking the database application program. Then the static semantic analyzer is used to check all statements that update data items in each transaction to find out the read, prewrite, and postwrite sets for each data item that is updated in the transaction. Other statements that are not for updating purpose are not checked.

Let's look at an example to see how to construct the read, prewrite, and postwrite sets by using the static semantic analyzer. Consider the following three

Table I. Result of Static Semantic Analysis After T_1 Is Analyzed

	Read set	Prewrite set	Postwrite Set
X	$\{\{a, b, c\}\}$	$\{\emptyset\}$	$\{\{y, z\}\}$
Y	$\{\{x, u\}\}$	$\{\{x\}\}$	$\{\emptyset\}$
Z	$\{\{x, w, v\}\}$	$\{\{x, y\}\}$	$\{\emptyset\}$

SQL statements in the given sequence appearing in one transaction, say T_1.

$$\begin{aligned}&Update\ Table1\ set\ x = a + b + c\ where \ldots\\&Update\ Table1\ set\ y = x + u\ where \ldots\\&Update\ Table1\ set\ z = x + w + v\ where \ldots\end{aligned}$$

In this example, x, y, z, etc., are used to represent attributes in a table instead of explicitly listing the attribute names. The result of static semantic analysis is as shown in Table I.

Suppose in another transaction, say T_2, the following statements are used to update x and w as follows:

$$\begin{aligned}&Update\ Table1\ set\ x = a + d\ where \ldots\\&Update\ Table1\ set\ w = x + c\ where \ldots\end{aligned}$$

The table constructed by the static semantic analyzer is used to check the transactions in the database log to find out whether they conform to the data dependency represented by the read, prewrite, and postwrite sets. Before a transaction updates a data item, all data items in at least one data item set of its read set must be read and all data items in at least one data item set of its prewrite set must be written by the same transaction. Then, after a transaction updates a data item, all data items in at least one data item set of its postwrite set also must be written by the same transaction. Please note that we also consider the sequence of the elements in the read, prewrite, and postwrite sets. By considering this kind of sequence imposed by the transaction program, it will be easier to detect intrusions in a stricter sense.

Now let's go through an example to illustrate the use of Table II. Consider the following transaction T_i, and each data item updated in this transaction has the

Table II. Result of Static Semantic Analysis After T_2 Is Analyzed

	Read set	Prewrite set	Postwrite set
x	$\{\{a, b, c\}, \{a, d\}\}$	$\{\emptyset\}$	$\{\{y, z\}, \{w\}\}$
y	$\{\{x, u\}\}$	$\{\{x\}\}$	$\{\emptyset\}$
z	$\{\{x, w, v\}\}$	$\{\{x, y\}\}$	$\{\emptyset\}$
w	$\{\{x, c\}\}$	$\{\{x\}\}$	$\{\emptyset\}$

corresponding read, prewrite, and postwrite sets as shown in Table II.

$$T_i : r[y], r[a], r[q], r[b], r[c], w[x], r[x], r[w], r[v], w[z], r[x], r[u], w[y], r[c], r[d], r[e].$$

First the transaction is scanned to see which data items were updated, then the read set, prewrite, and postwrite sets are checked. For data item x, the read set consists of two data item sets $\{a, b, c\}$ and $\{a, d\}$. By checking the transaction T_i it is found that before the operation $w[x]$ this transaction read data items $\{a, b, c\}$ but not $\{a, d\}$. As long as all data items in at least one data item set of the read set are read before updating one data item, the read set of this data item is satisfied. So in this case, the read set of x is satisfied. Then the prewrite set for x is checked. Since the prewrite set of x is $\{ø\}$, there is no need to check whether any particular data item is updated before $w[x]$. Hence the prewrite set of x is also satisfied. However, the postwrite set of x is not checked at this time, because to do that one must scan the log to the end of the transaction and it may take a long time if the transaction is large. Moreover, in order to do that one must wait until the transaction is committed; in that case, it is not possible to stop a malicious transaction before it's committed. So z, the next data item updated, is checked. The read set consists of one data item set $\{x, w, v\}$. Before the write operation $w[z]$ this transaction read data items $\{x, w, v\}$, therefore, the read set of z is satisfied. The prewrite set of z consists of one data item set $\{x, y\}$. It is found that before the operation $w[z]$, x is updated but y is not. So the prewrite set of z is not satisfied. After the transaction executes $w[z]$, our database IDS could detect an anomaly in this transaction and notify the site security officer. Thus, the malicious transaction can be stopped by rolling back the transaction. In this case, the modification has not been reflected in permanent storage, so no data are damaged.

It must be noted that our proposed method is not able to detect the malicious transactions that are compliant to the data dependencies observed in normal user transactions. In the case that a data item that is not dependent on any other data items is updated by a malicious transaction, only employing data dependencies is not enough for detecting it.

4.2. Dynamic Semantic Analyzer

After the analysis of the static semantic analyzer, there may be multiple data item sets in the read set, prewrite set, or postwrite set for one data item. In the practical use, some sets are more frequently used than others. This depends on the execution path of the database application and the normal user access pattern of the database. The access probability for these sets can be used to identify malicious

transactions. The dynamic semantic analyzer can determine the *use probability* for each of these sets. We define the use probability in both static and dynamic analysis cases below.

- *Static Use Probability*, P_s, for one data item set in read, prewrite, or postwrite set is defined as: $P_s = 1/T_s$, where T_s is the total number of data item sets in the read, prewrite, or postwrite set. P_s implies that all data item sets in the read, prewrite, or postwrite set will be equally likely referred. For example, in Table II, data item x has two data item sets in its read set, i.e., $\{a, b, c\}$ and $\{a, d\}$. So $T_s = 2$, and P_s for $\{a, b, c\}$ is $1/2 = 50\%$, and also the same for the set $\{a, d\}$.
- *Dynamic Use Probability*, P_d, for one data item set in read, prewrite, or postwrite set is defined as: $P_d = n/k$, where n is the number of times each data item set in the read, prewrite, or postwrite set is referred in the history and k is the total number of times the data item is updated in the history. For example, if data item x is updated 10 times in the history, then $k = 10$. By checking all transactions in the history which updated x, we can find how many times each data item set in the read, prewrite, or postwrite set is used and get the number n. Then, the dynamic use probability for x can be computed. It must be noted that the dynamic use probability should be recalculated frequently to reflect the current characteristics of data dependency.

The *total use probability*, P_{total}, for one data item set in read, prewrite, or postwrite set will be $P_{total} = P_s \times \text{weight} + P_d \times (1 - \text{weight})$ where *weight* is a number between 0 and 1 which is decided by the Site Security Officer.

The total use probability for one data item set can be used to identify some infrequently used data item sets. The reason total use probability is the weighted sum of static use probability and dynamic use probability is that sometimes the history or training data may not be enough to reflect the real dynamic use probability of them. This can be achieved by specifing a threshold $P_{threshold}$ for the total use probability. If $P_{total} < P_{threshold}$, then this data item set will be tagged as infrequently used set.

For example, in Table II, the read set of data item x contains two sets, *S1:* $\{a, b, c\}$ and *S2:* $\{a, d\}$. If the static semantic analyzer assumes they are equally likely to have been used by the operation on the database, then for *S1*, we have $P_s = 50\%$. Suppose that in practicality *S1* is never read. That means for *S1* the dynamic use probability $P_d = 0\%$. If the weight is 30%, the total use probability $P_{total} = 50\% \times 30\% + 0\% \times (1 - 30\%) = 15\%$. If $P_{threshold}$ is set to 20%, then $P_{total} < P_{threshold}$, so we can identify the data item set $\{a, b, c\}$ as infrequently used data item set of the read set of data item x. In the case when a user transaction reads the set $\{a, b, c\}$ for updating x the system will indicate an anomaly.

5. DATA DEPENDENCY AT THE USER TASK LEVEL

As we illustrated above, by checking the read, prewrite, and postwrite sets for data items updated in one particular transaction, many anomalous activities performed by the transaction can be detected. But it's hard to find the anomalous activity carried out by a group of transactions, each of which satisfies the read, prewrite, and postwrite sets for the data items updated in the course of the activities. We propose a method below to identify the anomaly based on data dependency among transactions. The idea is that when a user executes some part of the database application to fulfill the user task, generally not one but several transactions are submitted to the DBMS. For instance, when a customer transfers money from one account to another account, there may involve several transactions. One transaction may be used for reducing the balance of one account by some amount and increasing the balance of another account by the same amount. Another transaction may be utilized to update some internal accounts only for the use of the bank, e.g., some accounts used for statistical or audit purpose. The number of transactions used for one user task is limited. Also, the possible execution sequences of these transactions are decided by the database application. By using a training procedure, we try to find out what data items are updated and also determine their update sequence in a user task, e.g., a deposit task in a banking application.

In the training phase, a user task is executed extensively to make sure almost all different cases for this user task are performed. Since these executions must also satisfy the read, prewrite, and postwrite sets of data items updated, we check which data item set in the read, prewrite, and postwrite sets is actually used in the training phase. Then, we use the actual postwrite set and read set for creating the data dependency among transactions. Based on the definition of postwrite set, it's natural to use postwrite set to construct the normal data update sequence. The reason we use the read set besides using the postwrite set is as follows. Suppose data item x is updated then data item x, y, z should be updated consequently. And after data item y is updated, data item u and v also should be updated. The situation is x, y, z, u, and v are not necessarily updated in the same transaction. It's possible that x, y, and z are updated in one transaction, and then u and v are updated in another transaction. Even it is possible that x, y, and z are not updated in the same transaction. For example, the following transactions T_1 and T_2 are executed consecutively in a user task:

T_1: *update Table1 set* $x = 120\%\ x$ *where ...*
update Table2 set $y = x + a$ *where ...*
update Table3 set $z = x + b$ *where...*

T_2: *update Table4 set* $u = y + c$ *where...*
update Table5 set $v = y + d$ *where...*

Table III. The Read, Prewrite, and Postwrite Sets for T_1

	Read set	Prewrite set	Postwrite set
x	$\{\{x\}\}$	$\{ø\}$	$\{\{y, z\}\}$
y	$\{\{x, a\}\}$	$\{\{x\}\}$	$\{ø\}$
z	$\{\{x, b\}\}$	$\{\{x, y\}\}$	$\{ø\}$
u	$\{\{y, c\}\}$	$\{ø\}$	$\{ø\}$
v	$\{\{y, d\}\}$	$\{ø\}$	$\{ø\}$

The read, prewrite, and postwrite sets constructed based on these two transactions are shown in Table III.

From Table III, it can be seen that the postwrite set for data item y is $\{ø\}$; that means after updating y no other data items need to be updated by the same transaction. And since the prewrite set for u and v is $\{ø\}$, before the transaction updates u and v, nothing needs to be written by the same transaction. However, since T_1 and T_2 always execute consecutively, after y is updated, u and v must be updated based on the new value of y. So at the higher level of the user task instead of the transaction, we can find some new data dependencies.

5.1. Write-Chain

To help understand write-chain, the notions of *active* and *passive* data items are defined as follows.

Definition 4. An *active* data item x is the data item that causes other data item(s) to be updated when x itself is updated.

Definition 5. A *passive* data item is the data item that is updated as a result of other data item(s) being updated.

Based on the above discussion, we define a term *write-chain* to capture this kind of data dependency at the user task level. *Write-chain* is a sequence of data items that are always updated together and have complete order among these data items. The first data item of the write-chain is the active data item; other data items are passive data items. It's clear that some write-chains can be created directly from the postwrite set of data items. For example, from Table III we can have write-chain $wc1 : x \rightarrow y \rightarrow z$ based on the postwrite set of x. Other write-chains are deduced from the data dependencies among y, u, and v that are hidden in several transactions.

The method for constructing write-chains deduced from several transactions is as follows. Suppose there's a write operation $w[y]$ in one user task which includes several transactions. Check the $rs(y)$ to see if there is any data item x_i, $x_i \in$ any data item set in $rs(y)$, updated before $w[y]$ in this user task. And if there's anyone, create a write-chain wc: $x_i \rightarrow y$. Still using the above example, from T_2 we get

$y, c \in rs(u)$. Moreover, y is updated in T_1 before write operation $w[u]$ in T_2. So the write-chain $y \to u$ can be deduced. Similarly, another write-chain $y \to v$ can also be obtained from the above example.

By considering the data update sequence in the transactions, we can combine several write-chains into one. For example, in transaction T_2, u is updated before v and both of them are updated because y is updated. So we can come up with a combined write-chain $y \to u \to v$.

5.2. Data Update Dependency Graph

After constructing write-chains for one user's task from the read set and postwrite set, it may be observed that there are some common data elements among these write-chains. It will be useful to connect these write-chains to a graph-like structure to reflect the data dependency among transactions. We call this structure Data Update Dependency Graph (DUDG). For example, consider the following five write-chains.

$$\begin{aligned}
&wc1 : x1 \to x2 \to x3 \to x4\\
&wc2 : x2 \to x5 \to x6\\
&wc3 : x5 \to x7\\
&wc4 : x3 \to x8\\
&wc5 : x6 \to x9 \to x10
\end{aligned}$$

The first data item at each write-chain is the active data item. All other data items following it are passive data items. This means all these passive data items are updated in this user task because the active data item is updated. It is assumed that a data item is updated only once in a user task. This assumption will make it easier to find data dependencies among transactions. Although theoretically a user task may update one data item more than once, this case is rare in practice.

The normal write sequence for one user task is decided by different execution paths under different situations. One or several DUDGs can be constructed to reflect all possible write-chains and their correlations for one user task. In Fig. 1, we construct one DUDG to represent the above example. In Fig. 1, a solid arrow from A to B represents A as the active node and B as the passive node meaning B is updated because A is updated, i.e., $B = f(A)$. Whereas a dashed arrow from A to B represents that both A and B are passive nodes and $B \neq f(A)$. It is needed to trace back from B to find the first solid arrow. The element at the tail of the solid arrow is the active node, which causes B to be updated.

The DUDG graph models the partial order in which data items are updated in a user task. For example, in Fig. 1 if *x1* is updated, all other data items must be updated in this user task. Furthermore, all these data items should be updated according to the partial order in this DUDG graph. The DUDG graph profiles

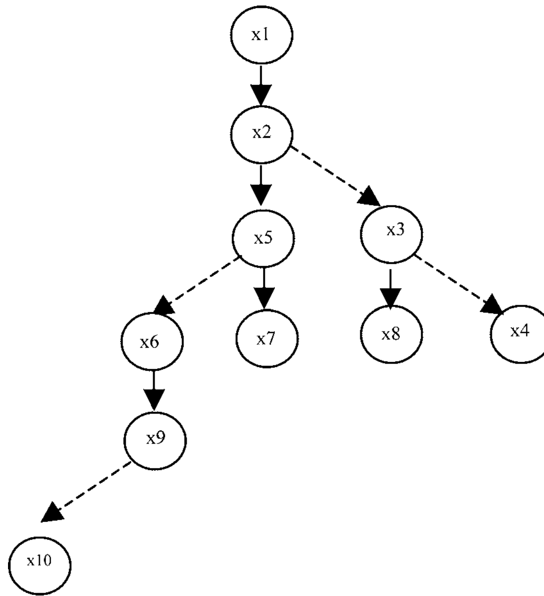

Fig. 1. Data update dependency graph.

the normal data update sequence for a user task and can be used for detecting anomalous transactions in the database system.

6. PETRI-NET IMPLEMENTATION OF THE DATA DEPENDENCY MODEL

Petri-Net is a modeling tool to specify systems that are concurrent, distributed, parallel, nondeterministic, and asynchronous. A Petri-Net is a bipartite directed graph, which consists of two kinds of nodes, namely places and transitions. Places are represented by circles and transitions are represented by bars. The edges are between transitions and places, and they indicate the input or output to the transitions. Each place can hold tokens and only when the input places for a transition has required tokens, the transition can fire. Due to space constraint we do not provide detailed information about Petri-Nets other than what is needed for our implementation. Interested readers may study the article by Murata [17].

6.1. Petri-Net Implementation of DUDG

In this work, we use Petri-Nets to model data relationships, particularly the DUDG. Some researchers have also used Petri-Nets for multisource attacks

detection [18]. However, our model is different from theirs as can be observed below. In Fig. 2, each place represents the write operation for one data item in the DUDG. We use the name of the data item to represent each node. The transition represents the end of the input operations and beginning of output operations. An additional place named “end” is added to identify the normal write sequence. Additionally, we use two colors for tokens, red and blue.

The tokens with different colors are used to guarantee that a transition occurs only when the write sequence in the database log is in the required sequence. This was different from the general transition rule of Petri-Nets. When a write operation is found in the database log, a blue token is added to the place identified by the name of the data item. When a transition fires, a red token is put to each output place of the transition. Suppose a place x_i is an output place of a transition t_j. If a place x_i holds a blue token, then a transition t_j fires which adds a red token to $x_{i,}$ that causes misfire. The newly added red token is removed from x_i and the blue token(s) from the input place(s) of t_j are also removed. Whenever this happens, we can infer that a write operation corresponding to the place x_i is executed before the execution of write operation(s) that caused the transition t_j to fire. This indicates that this write sequence doesn’t conform to normal update sequence.

6.2. Modeling Time Pattern of Transaction Execution Over the Petri-Net

A reasonable range in the database log needs to be decided to check the Petri-Net model. If we can find out a method to group transactions in the database log that are used collectively to perform one user task, then that user task can be tested using the Petri-Net. The time pattern of transaction executions can be used for this purpose.

In order to facilitate our discussion, we define the term *gap*, which is used to describe the time difference between the two consecutive user tasks or two consecutive transactions submitted by the same user in the database log. The gap refers to time difference between the end of one task (transaction) and the beginning of another task (transaction). In the following discussion, the *gap between two user tasks* refers to the gap between two consecutive tasks submitted by the same user. Similarly, the *gap between two transactions* refers to the gap between two consecutive transactions submitted by the same user.

Generally, the gap between two user tasks is much bigger than the gap between two transactions. For example, when the accountant in the bank serves customers, there’s always a time interval from several seconds to several minutes between serving two different customers. This time interval is the gap between two user tasks. In the case the machine load is not very heavy, we can always assume this interval is bigger than the interval between executions of two transactions in a user task. By checking the distribution of the length of the gap between two transactions, grouping transactions in the database log into user tasks becomes

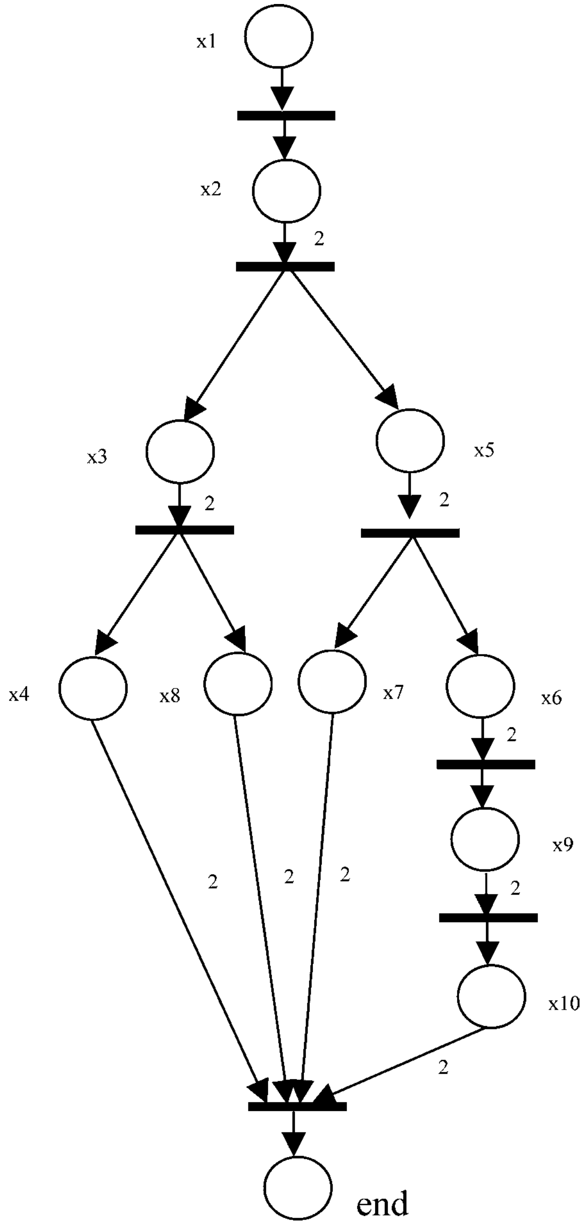

Fig. 2. Petri-Net implementation of DUDG.

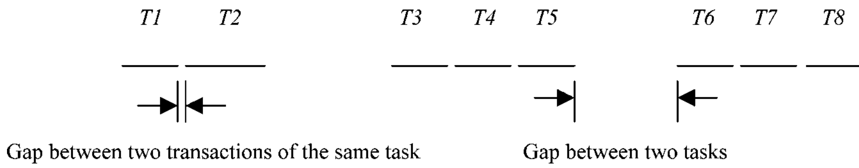

Fig. 3. Transaction and task gaps.

simple. The timestamps indicating the end of one transaction and beginning of another can be used to calculate the length of the gap between the two transactions.

In Fig. 3, we illustrate the gap between two transactions of the same task and the gap between two tasks. A threshold is used to decide the gap between two tasks. Whenever any two transactions' gap is larger than the threshold, we consider the end of the first transaction as the end of a user task. That means the next transaction after the gap does not belong to the same user task.

By using the method described above, the gap between two user tasks can be determined. All the transactions between two of these consecutive gaps can be grouped together and considered to be the part of the same user task. Although the purpose of the user task is unknown, the Petri-Net modeled normal updated sequence can be applied to this user task to check whether there are anomalies in the task.

To do so, the data items that are updated in the transactions of the user task are checked. The transactions of the user task are scanned and whenever a write operation is found, all Petri-Nets are checked to see if any of them has a place corresponding to the name of the data item updated. If there is one, a blue token is put into that place. It is possible that several Petri-Nets may have a place corresponding to the name of the data item updated. In that case, a blue token is added to each of those places of those Petri-Nets. After the last write operation in the user task is checked, the Petri-Nets are assessed to see if any of them have all the transitions fired and there's a red token in the "end" place. If there's at least one Petri-Net having a red token in the "end" place, this user task is considered to be a normal user task. Otherwise, the user task is considered anomalous indicating these transactions as malicious.

7. EXPERIMENT

A simulation model was developed in order to evaluate the performance of the proposed database intrusion detection approach. We pursued experiments on both transaction levels and user task level intrusion detections. The performance of the proposed database intrusion detection model relies on several factors. The first factor is to what extent the data in the database are dependent on each other.

The stronger the dependency, the better the performance. The test result shows that this model is especially useful in the environment where mathematical operations are often involved in database transactions. The second factor is the setting of operating parameters of the database intrusion detection system. For example, the SSO needs to set the weight and threshold parameters for the dynamic semantic analysis. The performance of the system will degrade if these parameters are incorrectly set up.

7.1. Experimental Model

The modules of the simulation system can be roughly divided into three main components: transaction level data dependency analysis, user task level data dependency analysis, and detection. The transaction level data dependency analysis is responsible for analyzing the source code of the normal database transactions and database logs for generating read set, prewrite set, and post-write set. The user task level data dependency analysis is exploited to generate the write chain and Petri-Net. These structures are used to model the normal data update pattern at the user task level. The detection part accepts the output of these previous two components and generates alarms when any malicious transaction is detected.

The detailed modules of each component and their relationships are illustrated in Fig. 4. An arrow connecting two modules indicates the direction of information flow.

7.2. Experiment Design

To evaluate the effectiveness of the proposed detection methods at different levels, we employ three different experiment settings. The first one only uses the results of static semantic analyzer to detect intrusion in the database log. The second one utilizes both static semantic analyzer and the dynamic semantic analyzer's results to detect intrusions. The last one employs user task level intrusion detection method as well as transaction level intrusion detection method. Performances of these three experiments are compared to illustrate the effectiveness of each of these methods.

It's noted that there are a number of parameters that may affect the performance of the model. In order to evaluate the sensitivity of the performance to some parameters of the system, we adopt the following methods. First, a baseline setting of system parameters is presented in Table IV. These parameters are observed to be the typical setting of some normal user environment. Then, by varying one parameter at a time, we evaluate how the intrusion detection system responds to the change of the parameter and whether the performance is sensitive to this change. This also facilitates evaluating different detection methods proposed.

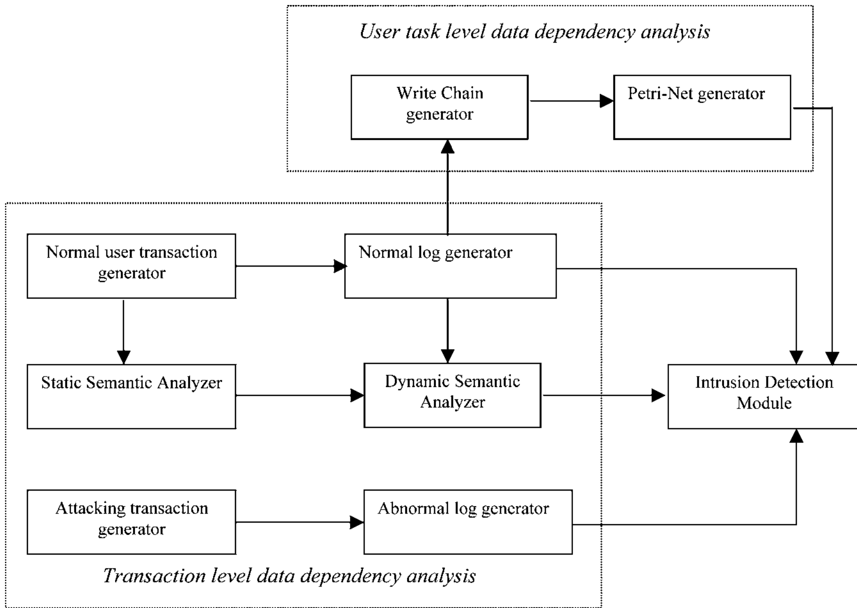

Fig. 4. Model of the experiment.

Table IV. Parameters Used in Test Settings

Parameter	Range	Baseline
Transaction level parameters—normal user transaction		
Average number of data items read in each update statement	1–5	2
Average number of data items updated in each transaction	1–10	2
Number of transactions generated	100–1000	100
Average number of common data updated by test transactions	10–20	10
Transaction level parameters—intruding transactions		
Average number of data items read in each update statement	1–10	2
Average number of data items updated in each transaction	1–20	2
Number of transactions generated	100–1000	100
Average number of data updated by test transactions	10–40	10
User task level parameters		
Average number of transactions in each normal user task	1–10	10
Average number of transactions in each malicious user task	1–20	10
System parameters set by SSO		
Weight used in dynamic semantic analysis	20%	20%
Threshold used in dynamic semantic analysis	15%	15%

7.3. Performance Analysis

The experiment results on hit rate and miss rate of the detection are collected for performance analysis. The hit rate consists of the true positive rate and true negative rate. The miss rate consists of the false positive rate and false negative rate. The true positive rate and true negative rate show the percentage of the correct identification of attacking and normal transactions, respectively. While the false positive rate and false negative rate give the percentage of the incorrect identification of attacking and normal transactions, respectively. We present each simulation result and the explanation of it as follows. In each of the following figures, we only show true positive rate and false positive rate for one test setting. The false negative rate and true negative rate are not illustrated. The reason is that the false negative rate is equal to 100% minus true positive rate and, similarly, true negative rate is equal to 100% minus false positive rate.

First, the performance of detecting malicious transactions based only on static semantic analysis is tested. The average number of write operations in a transaction is varied from one to five. All other parameters in the system are kept intact. It can be seen in Fig. 5 that when the average number of write operations in a transaction is one, the true positive rate is as low as 44%. In this case, only the read set is used for detecting intrusions. With the increasing number of write operations, it's noted that the detection rate climbs up very quickly. When the average number of write operations is three, we can get satisfactory true positive rate, which is 89%. The increased true positive rate is due to more data dependencies observed which are modeled by the prewrite sets and postwrite sets.

Then, we test to what extent the increased number of read operations in a SQL statement affects the performance of detection based on static semantic analysis. The average number of read operations in an SQL statement is changed from one to five in our test. In Fig. 6, the experiment result is presented. When on average there's only one data item read before an update SQL statement modifies a data

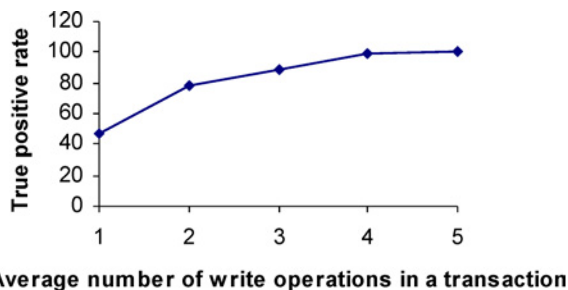

Fig. 5. Performance of detection based on static semantic analysis with varied number of write operations in a transaction.

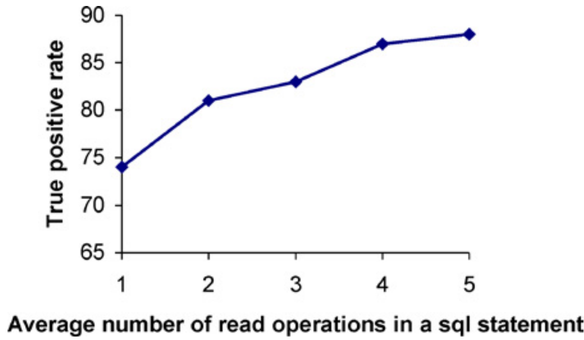

Fig. 6. Performance of detection based on static semantic analysis with varied number of read operations in a SQL statement.

item, the true positive rate is at 74%. The best performance is achieved at 88% true positive rate when the average number of read operations in an SQL statement is five. The difference between these two rates is 14%. Comparing this result with Fig. 5, it's observed that the true positive detection rate is more sensitive to the average number of update statements in a transaction. That means if there are more update statements per transaction, the detection rate may increase quickly provided other parameters are kept intact.

Figure 7 presents a comparison of detection performances based on static and dynamic semantic analyses on the same setting of test data. The threshold for dynamic semantic analysis is set to 15%. It means that if the probability of a data item set from the read set (or prewrite set, or postwrite set) being used is less than 15%, it will be tagged as infrequently used set. So it's discarded from

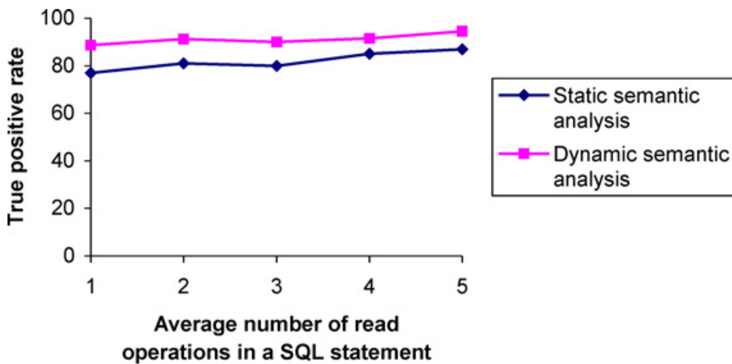

Fig. 7. Comparison of detection performances based on dynamic semantic analysis and static semantic analysis.

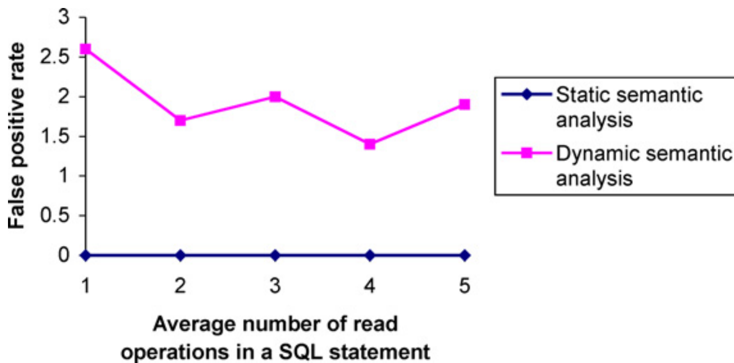

Fig. 8. Comparison of false positive rate between dynamic semantic analysis and static semantic analysis.

the corresponding read set (or prewrite set, or postwrite set). In our test settings, the true positive rate of dynamic semantic analysis improved by almost 6–12% compared to that in the static semantic analysis. It proves that when constructing transaction level data dependencies by considering the normal data update pattern in the history or training data, the detection becomes more accurate.

Figure 8 illustrates that when using the detection method based on the dynamic semantic analysis the false positive rate increases very little, i.e., by about 1.5–2.6%. Since the true positive rate increases by about 6–12% as illustrated in Fig. 7, so the overall performance of dynamic semantic analysis achieves the desired better performance.

Next, we illustrate the performance of the simulation system at detecting malicious user task. Figure 9 shows the true positive rate for identifying malicious

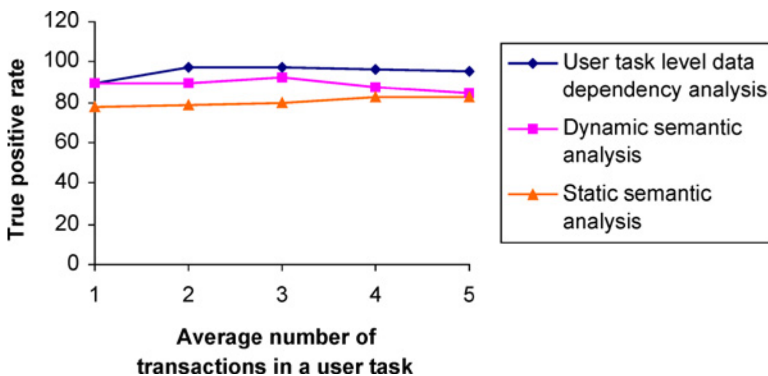

Fig. 9. Comparison of true positive rates observed at different detection levels.

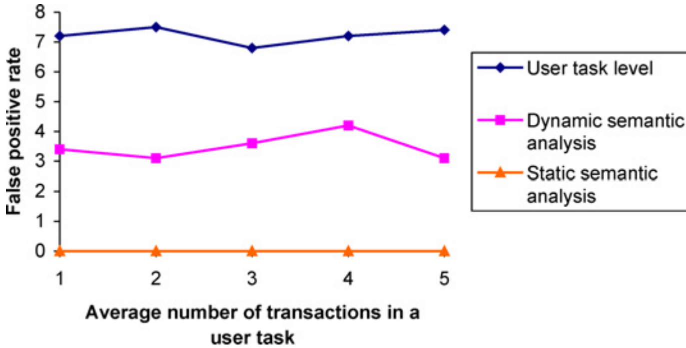

Fig. 10. Comparison of false positive rate observed at different detection levels.

transactions in user tasks. It's observed that at our test setting, the user task level true positive rate is about 5–10% better than only using the transaction level detection methods. By using detection methods at both levels, we can achieve desirable performance that makes the true positive rate to be about 90–97%. The test also shows that the false positive rate doesn't increase much. It can be seen in Fig. 10 that the false positive rate increases about 3–4% compared to the dynamic semantic analysis. This is still acceptable and the overall performance is better when the user task level intrusion detection method is used.

8. CONCLUSIONS

We have offered a database intrusion detection model that uses data dependency relationships in user tasks or transactions. Dependencies are determined by using the read, prewrite, and postwrite sets of data items, which are generated by the static and dynamic semantic analyzers. User applications or database logs can be checked to construct these sets. By finding the data dependencies among transactions, we identify anomalies hidden at the user task level. A Petri-Net based implementation concept has been offered to check these kind of data correlations at user task level as opposed to the transaction level. Moreover, time pattern of transaction executions is used to identify the range of database log for verifying normal data update sequence in order to reduce false negatives. The simulation results illustrate that by combining the transaction level and user task level intrusion detection methods better performance can be achieved.

ACKNOWLEDGMENT

We are thankful to Dr. Robert L. Herklotz for his support, which made this work possible. This work was supported in part by US AFOSR under grant No. F49620-01-10346.

REFERENCES

1. B. Panda and J. Giordano, Defensive information warfare, *Communications of the ACM*, Vol. 42, No. 7, pp. 31–32, July 1999.
2. Liu, P. Ammann, and S. Jajodia, Rewriting histories: Recovering from malicious transactions, *Distributed and Parallel Databases*, Vol. 18, No. 1, pp. 7–40, January 2000.
3. R. Sobhan and B. Panda, Reorganization of database log for information warfare data recovery, *Proceedings of the 15th Annual IFIP WG 11.3 Working Conference on Database and Application Security*, July 2001.
4. J. Zhou, B. Panda, and Y. Hu, Succinct and fast accessible data structures for database damage assessment, *Proceedings of the International Conference on the Distributed Computing and Internet Technology*, December 2004.
5. H. S. Javitz and A. Valdes, The SRI IDES Statistical Anomaly Detector, *Proceedings of the IEEE Symposium on Security and Privacy*, May 1991.
6. T. F. Lunt, R. Jagannathan, *et.al.*, IDES: A progress report, *Proceedings of the 6th Annual Computer Security Applications Conference*, December 1990.
7. S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, A sense of self for Unix processes, *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, IEEE Computer Society, pp. 120–128, 1996.
8. A. K. Ghosh, A. Schwartzbard, and M. Schatz, Learning program behavior profiles for intrusion detection, *1st USENIX Workshop on Intrusion Detection and Network Monitoring*, 1999.
9. T. Lane and C. E. Brodley, Sequence matching and learning in anomaly detection for computer security, *Proceedings of the AAAI-97 Workshop on AI Approaches to Fraud Detection and Risk Management*, pp. 43–49, 1997.
10. J. Frank, Artificial intelligence and intrusion detection: Current and future directions, *Proceedings of the 17th National Computer Security Conference*, October 1994.
11. W. Lee and S. Stolfo, Data mining approaches for intrusion detection, *USENIX Security Symposium*, 1998.
12. W. Lee, R. A. Nimbalkar, K. K. Yee, S. B. Patil, P. H. Desai, T. T. Tran, and S. J. Stolfo, A data mining and CIDF-based approach for detecting novel and distributed intrusions, *Proceedings of the 3rd International Workshop on the Recent Advances in Intrusion Detection*, October 2000.
13. Y. Huang, W. Fan, W. Lee, and P. Yu, Cross-feature analysis for detecting ad-hoc routing anomalies, *Proceedings of the 23rd International Conference on Distributed Computing Systems*, May 2003.
14. V. C. S. Lee, J. A. Stankovic, and S. H. Son, Intrusion detection in real-time database systems via time signatures, *Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium*, 2000.
15. C. Chung, M. Gertz, and K. Levitt, DEMIDS: A misuse detection system for database systems, *Third Annual IFIP TC-11 WG 11.5 Working Conference on Integrity and Internal Control in Information Systems*, Kluwer Academic, pp. 159–178, November 1999.
16. E. Codd, A relational model for large shared databanks, *Communications of the ACM*, Vol. 13, No. 6, pp. 377–387, June 1970.
17. T. Murata, Petri-Nets: Properties, analysis, and applications, *Proceedings of the IEEE*, Vol. 77, No. 4, pp. 541–580, April 1989.
18. B. Panda and R. Yalamanchili, A host-based multisource information attack detection model design and implementation, *Information: An International Journal*, Vol. 4, No. 4, October 2001.

Yi Hu is a PhD candidate in Computer Science and Computer Engineering Department at the University of Arkansas. His research interests are in Database Intrusion Detection, Database Damage Assessment, Data Mining, and Trust Management. Previously, he received the BS and MS degree in Computer Science from the Southwest Jiaotong University and the University of Arkansas, respectively.

Brajendra Panda received his MS degree in mathematics from Utkal University, India, in 1985 and PhD degree in computer science from North Dakota State University in 1994. He is currently an associate professor with the Computer Science and Computer Engineering Department at the University of Arkansas. His research interests include database systems, computer security, digital forensics, and information assurance. He has published over 60 research papers in these areas.