Defensive Execution of Transactional Processes against Attacks

Meng Yu, Wanyu Zang
Department of Computer Science
Monmouth University, 07764
myu@monmouth.edu

Peng Liu
School of Information Sciences and Technology
Pennsylvania State University, 16801
pliu@ist.psu.edu

Abstract

It is a well known problem that the attack recovery of a self-healing system rolls back not only malicious transactions, but also legitimate transactions that are dependent on the malicious transactions. Rolling back and re-executing damaged transactions increase the response time of the system and may cause a significant processing delay. In such situations, the availability of the system is compromised and the system suffers the vulnerability of Denial of Service (DoS). In this paper, we propose a defensive executing technique and analyze its effectiveness. Our technique concurrently executes multiple paths of a transactional processes based on the prediction generated by a Discrete Time Markov Chain. The defensive execution can reduce the delay caused by recovery. We also propose a branch cutting technique to reduce the extra cost introduced by defensive execution. Our analytical results show that our technique is practical against transactional level attacks.

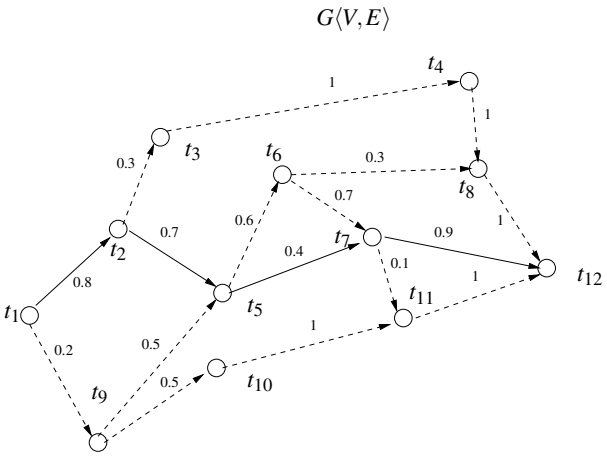

Figure 1. An example of transactional processes

1 Introduction

Distributed transactional processing systems (e.g., distributed database systems and workflow systems) are important in most critical infrastructures such as financial services. These services rely on the correctness, availability, and reliability of the processing systems. Each transactional process consists of a set of transactions that are related to each other in terms of the semantics of a business process. Each transaction represents a specific unit of work that the business needs to do (e.g., a specific application program, a database transaction). A consistent and reliable execution of distributed transactional processes is crucial for all organizations.

However, it is well known that system vulnerabilities cannot be totally eliminated, and such vulnerabilities can be exploited by attackers who penetrate the system. Even worse, in such situation, legitimate transactions referring to the damaged transactions will also be affected since they compute results or make decisions based on wrong information.

Figure 1 shows an example of damage spreading and repair. In the figure, nodes are transactions and edges indicate executing paths. The destination node of a directed edge starts after the source node is complete, e.g., t_{12} will start after t_7 is done. If a node has more than one successors, the number on an edge indicates the probability that the specific path (or node) can be taken (Section 3 discusses how to determine the probabilities). In the figure, continuous lines are the actual executing path and dash lines are possible executing paths.

In Figure 1, path $t_1t_2t_5t_7t_{12}$ has been executed. If t_2 was compromised by an attacker, it would be possible that t_2 made a wrong decision to select t_5 as its successor. t_7 may also generate wrong information based on t_2's results. The example shows a single compromised transaction may affects many legitimate transactions. If we want to recover the transactional process and given t_{12} is correct anyway (t_{12} does not depend on any of t_2, t_5 and t_7) we need to roll back

(undo) $t_2t_5t_7$ and redo t_2 to select correct executing path. In the procedure, damage tracing and repair are involved.

We have to rely on attack recovery techniques to address the above problem. After the attacker successfully beats prevention techniques, attack recovery techniques restore the integrity level by repairing the damage caused by successful intrusions.

Previous work [1, 10, 14, 15] introduced different techniques to trace damage spreading and repair the damage in transactional processing systems. However, if attacks happen, all affected transactions will be rolled back (undone) and redone.

Rolling back and re-executing damaged transactions increase the response time of the system and may cause a significant processing delay. In such situations, the availability of the system is compromised and the system suffers the vulnerability of Denial of Service (DoS).

In this paper, we propose a defensive executing technique and analyze its effectiveness. Our technique concurrently executes multiple paths of a transactional processes based on the prediction generated by a Discrete Time Markov Chain. The defensive execution can reduce the delay caused by recovery. We also propose a branch cutting technique to reduce the extra cost introduced by defensive execution. Our analytical results show that our technique is practical against transactional level attacks.

This paper is organized as follows. Section 2 defines important terms used in this paper. Section 3 describes the details of our defensive executing technique. Garbage collection will be discussed in Section 4. We discussed the effectiveness of our technique in Section 5 and compare it with related work in Section 6. Finally, Section 7 concludes the paper.

2 Preliminary

In this section, we introduce some preliminary terms and concepts used in this paper.

2.1 Dependency relations

Dependency relations are important because not only data items are calculated through dependency relations, but also executing orders of concurrent transactions are determined by dependency relations. Furthermore, we need the dependency relations to determine proper defensive executions.

In Figure 1, the start node of an arrow will be executed right before the end node, which defines a *precedence* relation. For example, t_1 *precedes* t_2, which is denoted by $t_1 \prec t_2$. There are several possible executing paths in Figure 1. One of them is $t_1t_2t_5t_7t_{12}$. An other one is $t_1t_2t_3t_4t_8t_{12}$. t_2 makes a decision on which executing path will be selected. The selection between t_2's successor t_3 and t_5 is called *control dependence*, which is denoted by $t_2 \rightarrow_c t_3$ and $t_2 \rightarrow_c t_5$. In the figure, execution path $t_1t_2t_5$ was selected.

Assume $\prec$ is a relation on set $\mathscr{S}$ then we define $\text{minimal}(\mathscr{S},\prec) = \{x \mid x \in \mathscr{S} \wedge \nexists x' \in \mathscr{S}, x' \prec x\}$. Note there may be more than one result qualified by the definition of $\text{minimal}(\mathscr{S},\prec)$. For example, In Figure 1, given $\mathscr{S} = \{t_1,t_2,t_9\}$, since $t_1 \prec t_2$ and $t_1 prec t_9$, t_1 is the qualified results for $\text{minimal}(\mathscr{S},\prec)$. $\text{minimal}(\mathscr{S},\prec)$ defines start transactions in all left transactions of a transactional process.

Once t_2 is identified as a transaction compromised by the attacker, and the selection of execution path $t_1t_2t_5t_7$ was wrong, the execution of t_5 and t_7 need to be recovered. Furthermore, if t_{12} reads information generated by t_5, which is *flow data dependence* denoted by $t_5 \rightarrow_f t_{12}$, the damage will be spread to t_{12}. Note that even though no message has been sent from t_5 to t_{12}, they may still have data dependencies since they may share data objects.

In the above example, damage is spread through dependency relations. We can similarly define the other two types of data dependence. If t_j modifies data objects after t_i reads them, then t_j is *anti-flow dependent* on t_i, which is denoted by $t_i \rightarrow_a t_j$. If $t_i \prec t_j$, and they have common data objects to modify, then t_j is *output dependent* on t_i, which is denoted by $t_i \rightarrow_o t_j$.

All the relations $\rightarrow_f$, $\rightarrow_a$ and $\rightarrow_o$ are data dependency relations and are not transitive. From the well known results of concurrency and parallel computing, if t_j is data dependent on t_i, then they cannot run in parallel or concurrently, and $t_i \prec t_j$ must be satisfied. Otherwise, the value shared by t_i and t_j will be incorrectly calculated.

2.2 Unrecoverable transactions

In a distributed system, we need to consider both inside operations and interactions with the outside world. The formal models in previous work, such as [15, 14] were unable to formalize interactions with the outside world. Since interactions with the outside world are not recoverable, we use an *OWS* (outside world site) to model the outside world. All transactions happened on an *OWS* are *unrecoverable*. We consider all inputs obtained from users and all outputs to the users happen on a user site S_u which is a *OWS*. All user's transactions are called *OWT*s (outside world transactions). For example, a transaction that a user withdraws money from a ATM cannot be recovered. It is an *OWT* and the ATM is an *OWS*.

2.3 Transactional processes

With above notations, transactional processes can be modeled as $(T, S, \prec, \rightarrow_f, \rightarrow_a, \rightarrow_o, \rightarrow_c)$, where T is a set of transactions, S is a set of sites that are corresponding to a host or a processor in the distributed system, $S_u \in S$ is an OWS, and all dependency relations among transactions.

2.4 Concurrency restrictions and domino-effects

We use a simple example to explain that there do exist some restrictions on executing orders of transactions in dependency relation based recovery.

Consider transactions $t_1 : a = 1, t_2 : b = 2$, and $t_3 : y = a + b$, which are executed in the sequence of $t_1 \prec t_2 \prec t_3$. We have $t_1 \rightarrow_f t_3 \wedge t_2 \rightarrow_f t_3$. Assume that t_2 has been identified as compromised by an IDS, so the value of b is corrupted. Therefore, t_3 is also corrupted since it reads a incorrect b. During the concurrency restrictions, the rolling back and redoing damaged transactions have to strictly follow specific orders [15, 14]. The effects that a compromised transaction affects all legitimate transactions depending on it are called *Domino-effects*.

To recover, t_2 needs to be undone followed by redone. t_3 needs to be redone. Please note that t_3 does not need to be undone because in the example, no transaction is dependent on t_3. We must satisfy the sequence of undo$(t_2) \prec$ redo$(t_2) \prec$ redo(t_3) in the recovery. Any other execution will get wrong results. The precedence relations introduced by dependency relations is called *concurrency restrictions*.

The concurrency restriction is also caused by dependency relations. However, we can break anti-flow dependency relations by introducing multi-version data, as described in Section 3.

3 Defensive execution

This section describes the details of mathematical model based prediction, data structure support, and branch cutting in defensive executions.

3.1 A motivative example

Domino-effects always happen if transactional processes are attacked. During the recovery, all concurrency restrictions have to be strictly followed to guarantee that the recovery is correct [15, 14]. Both domino-effects and concurrency restrictions cause significant execution delay of recovery.

The recovery delay can be reduced or totally removed by defensive execution. The basic idea of defensive execution is to concurrently execute multiple paths that are most likely to happen. For example, in Figure 1, while executing path $t_1t_2t_5t_7t_{12}$ we also execute path $t_5t_6t_7$ and $t_3t_4t_8$ as *backup executions*. $t_1t_2t_5t_7t_{12}$ is called *actual execution*.

If the execution of path $t_1t_2t_5t_7t_{12}$ was manipulated by the attacker, and path $t_1t_2t_5t_6t_8t_{12}$ should be the correct execution, we can discard old execution and switch to the correct execution immediately, which may significantly reduce the delay of recovery. Please note that defensive execution does not handle the situation that the recovery path is the same to the attacked path, which can be handled by previous work [15, 14].

3.2 Discrete time Markov chain based self-adaptive prediction

Since executing all possible paths in a transactional process may cost too many resources, we use a mathematical model to choose the best candidate backup executions.

In a transactional process, if we consider the execution of each transaction as a state, and associate each transition from one state to an other, we will get a Discrete Time Markov Chain (DTMC) [13, 11]. We can predict the most possible executing paths in a transactional process according to its DTMC.

Given n transactions in a transactional process, the initial probability distribution is $\underline{\pi}(0) = (1, \underbrace{0, 0, \ldots, 0}_{n-1})$. After the k^{th} step, the probability that a specific transaction will be executed is given by

$$\underline{\pi}(n) = \underline{\pi}(0)\mathbb{P}^n \tag{1}$$

where $\mathbb{P}$ is the state-transition probability matrix of DTMC and $\mathbb{P} = \underbrace{\mathbb{P} \cdot \mathbb{P} \cdots \mathbb{P}}_{n}$. $\mathbb{P}$ is given by $\{a_{ij}\}$, where a_{ij} is the probability of executing t_j after transaction t_i is done.

The initial $\mathbb{P}$ can be determined in different ways. For example, it can be determined by the statistical results of executions, or provided by the designer of the transactional process. Otherwise, the initial $\mathbb{P}$ is determined in the following way. If t_i has m successors, we assign $\frac{1}{m}$ as the state-transition probability to each transition.

Once we have the initial $\mathbb{P}$, we revise it according to real executions. We record the actual executing path in each execution of the process, then revise $\mathbb{P}$ according to how frequently a successor will be selected in previous executions. Therefore, our DTMC based prediction is self-adaptive.

In Figure 1, we may obtain state-transition probabilities after enough executions, as associated numbers to each edge. For example, the edge from t_6 to t_7 is 0.7, which indicates that after t_6 is done, t_7 will have 70% chance to be taken. t_8 will have the other 30% change to be taken. The

state-transition probability matrix $\mathbb{P}$ is as follows.

$$\begin{pmatrix}
0 & 0.8 & 0 & 0 & 0 & 0 & 0 & 0 & 0.2 & 0 & 0 & 0 \\
0 & 0 & 0.3 & 0 & 0.7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0.6 & 0.4 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0.7 & 0.3 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.1 & 0.9 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0.5 & 0 & 0 & 0 & 0 & 0.5 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix} \tag{2}$$

a_{ij} in $\mathbb{P}$ is corresponding to the probability that transaction t_i transfers to transaction t_j.

We can use equation 1 to predict the execution of transactions. $(\underline{\pi}(0),\underline{\pi}(1),\underline{\pi}(2),\underline{\pi}(3),\underline{\pi}(4),\underline{\pi}(5),\underline{\pi}(6))^T$ will be

$$\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & .8 & 0 & 0 & 0 & 0 & 0 & 0 & .2 & 0 & 0 & 0 \\
0 & 0 & .24 & 0 & .66 & 0 & 0 & 0 & 0 & .10 & 0 & 0 \\
0 & 0 & 0 & .24 & 0 & .40 & .26 & 0 & 0 & 0 & .10 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0.28 & .36 & 0 & 0 & .02 & .34 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .03 & .97 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix} \tag{3}$$

where $\underline{\pi}(k), 0 \leq k \leq 6$ is the probability distribution in step k. For example, according to $\underline{\pi}(4)$, in the 4^{th} step, the probability to execute t_7 is 0.28 and the probability to execute t_8 is 0.36.

In the above results, if we select transactions with probability higher than 0.2 in each step, we will get transactions $t_1,t_2,t_3,t_5,t_4,t_6,t_7,t_8,t_{11},t_{12}$. These transactions cover exactly the paths $t_1t_2t_5t_7t_{12}$, $t_5t_6t_7$ and $t_3t_4t_8$ as we advocated paths at the beginning of Section 3. Practically, the threshold 0.2 in the example should be changed according to the applications.

The above example demonstrated how to use the DTMC model to predict the probability of executing a specific transaction in each step. Generally, in the k^{th} step of the execution of a transactional process, after the execution selected a successor t to execute, we select predicted transactions, other than t, with the highest probabilities in $\underline{\pi}(k)$ to execute. The probability of failing prediction will be pretty low according to the DTMC model.

So far we discussed node coverage in a transactional process. It will be a little more complicated to consider path coverage. Even though set $T = \{t_1,t_2,t_3,t_5,t_4,t_6,t_7,t_8,t_{11},t_{12}\}$ covers three different paths that we preferred, T also covers a path $t_5t_6t_8t_{12}$ in Figure 1. A problem is if we need to consider these paths separately.

Assume that there are a $x = 5$ in t_5, a $x = 3$ in t_6, and a $y = x+2$ in t_7. The path covering t_5t_7 and the path covering $t_5t_6t_7$ will be totally different. In such situation, the backup execution of t_7 should have a different version from the actual one. The reason that we need different versions of t_7 is that $t_5 \rightarrow_f t_7$ and $t_6 \rightarrow_f t_7$ and t_7 are covered by two different paths.

The following theorem serves as a general rule to determine if a specific transaction needs different executing version in a backup execution.

Theorem 1 *If a task t appears in both executing path P_1 and P_2 and one of P_1 and P_2 is a backup executing path, the execution of t will need separate versions for each executing path if and only if there exist $t_1 \in P_1$ and $t_2 \in P_2$ satisfying the following conditions:*

1. $t_1 \neq t_2$
2. $t_1 \notin P_2 \wedge t_2 \notin P_1$
3. $t_1 \rightarrow_f t \wedge t_2 \rightarrow_f t$

PROOF: If a task t appears in both executing path P_1 and P_2 and one of P_1 and P_2 is a backup executing path, the execution of t will need separate versions for each executing path if and only the executions of t in different paths generate different results.

The situation happens if and only if the computation of t refers to different data values on different paths, which can be described as $\exists t_1 \in P_1, t_1 \rightarrow_f t$ and $\exists t_2 \in P_2, t_2 \rightarrow_f t$. If $t_1 = t_2$ (two execution paths may have common part), t will generate the same results since the calculation is based on the same data set. Therefore, t_1 must be different from t_2. Otherwise, t cannot generate different results in two paths.

If $t_1 \in P_2$ or $t_2 \in P_1$, then t_1 or t_2 is in the common part of P_1 and P_2, there will be no difference for t in two executions P_1 and P_2. Therefore, we have $t_1 \notin P_2 \wedge t_2 \notin P_1$. □

3.3 Multiple revision histories to support multiple-branch execution

Defensive executions generate results from backup executions. The results will be useful if and only if user's executing path is discarded. Before the results become useful, they should be cached as temporary versions. We use multiple revision history to address the problem.

For any data object x written at time t_m, we associate t_m with x as its reversion number. We do not call it a timestamp since we do not perform recovery in real time. We assume that any two t_i and t_j, where $i \neq j$, are distinguishable in the system.

A revision history for data object x has a form $\langle x^{v_1}, x^{v_2}, \cdots, x^{v_n} \rangle$, where each $v_i, 1 \leq i \leq n$ is a revision number of x and v_j is later than v_i if $j > i$. If we know that x^{v_k} is corrupted by the attacker than any transaction that reads x^{v_k} get wrong results.

Please note that it is possible that in x's revision history, there is only a specific version that is corrupted. For example, x is generated periodically by a trustable transaction T and an attacker only corrupts a specific version of x, e.g.,

x^{v_k}. Therefore, we cannot conclude if x^{v_j}, where $j > k$, is dirty without further analysis.

For a specific version x^{v_k}, when it has a value that it is not supposed to have, it is *dirty*. For example, when x^{v_k} is created by an attacker or computed based on dirty data objects, it is dirty. Otherwise, it is *clean*.

A data object may have one *working history* corresponding to the actual executing path and multiple *caching history* corresponding to backup executions when necessary (see Theorem 1).

3.4 Operations on the revision history

A normal transaction reads data objects with the highest revision number, and it writes data objects with the highest revision number in their revision histories. So, a revision history does not change dependency relations among normal transactions. It operates just as if multiple versions did not exist.

A recovery transaction, whether it is an undo or redo transaction, operates on data objects with the same revision numbers as it used the first time it executed. For example, a undo(T_i) is implemented by removing all specific versions from revision histories of data objects written by T_i. A redo(T_i) will generate data objects with the same revision number as it executed first time. A revision history does not change dependency relations among recovery transactions either. We can consider that recovery transactions are for revising part of the history of the system.

When we find a dirty version x^{v_k}, there are two possible ways that the dirty version was generated. One possibility is that x^{v_k} should not exist at all, e.g, it was created by the attacker. Any transaction that reads x^{v_k} is supposed to read $x^{v_{k-1}}$ instead of x^{v_k}. Another possibility is that x^{v_k} has a dirty value and needs to be recomputed by a redo transaction. Any transaction that reads x^{v_k} needs to wait until the redo transaction has completed to get a correct value of x^{v_k}. In this case, we mark x^{v_k} as $x_b^{v_k}$ to block possible reading until the redo transaction is complete.

Multi-version data objects break dependency relations among recovery transactions and normal transactions, which enable us to run the recovery transactions and normal transactions concurrently. According to the structure of the revision history, operations on old versions happen as "in the past." Therefore, execution of normal transactions does not corrupt recovery transactions.

Please note that flow dependencies cannot be broken, which guarantees that the semantics of execution are correct. From the point of view of recovery transactions (or normal transactions), there is only a single version for each data object to ensure correct semantics.

By introducing multiple revision histories, the actual execution will work on the working history and all backup executions operate on caching histories. Once the actual execution is determined to be discarded, a caching history corresponding to the selected backup execution will be switched to the working history.

4 Branch cutting and garbage collection

Most IDSs have delays that are corresponding to their detection windows. After the detection window, IDSs usually do not check back. Then it will be safe to remove all results generated by backup executions to save computing resources. However, the IDS delay could be a random variable that subjects to a probability distribution. Furthermore, an IDS may detect intrusions not in a temporal order, which complicates the decision on the proper time to remove unnecessary results. We will address this problem in this section.

4.1 General cases

After a specific period, if the probability that a backup path will be used is low enough, we can remove the backup execution. We call such operation as *branch cutting*.

Proper branch cutting can benefit consequent execution greatly. For example, after a IDS detection window, if no intrusion has been detected in the path $t_1t_2t_5$ of the process in Figure 1, we can remove the backup execution t_2t_3. The branch cutting is marked as phase 1 in the figure. In the consequent execution, it is not necessary to run t_3t_4 as a back up execution again, which reduce the cost of defensive execution.

Similarly, after t_7 is done, the IDS may find no intrusions in the execution path $t_1t_2t_5t_7$. We can safely cut the backup executing branch $t_5t_6t_7$. Therefore, the backup execution $t_6t_8t_{12}$ will be not necessary in the following steps.

In the above example, the total extra cost of defensive execution were t_3 and t_6, which has been significantly reduced by branch cutting operations.

Garbage collection will be done whenever branch cutting happens. The garbage collection operation simply removes all revision histories generated by cut branches to save the storage.

Let us assume that the IDS delay is normally distributed with parameter T and σ (If the IDS delay has a different probability distribution, our following equations can be revised accordingly). The probability density function is as follows.

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-T)^2/2\sigma^2} \tag{4}$$

The expected value of IDS delay is

$$E(X) = \frac{1}{\sqrt{2\pi}\sigma} \int_{-\infty}^{\infty} e^{-(x-T)^2/2\sigma^2} dx = T \tag{5}$$

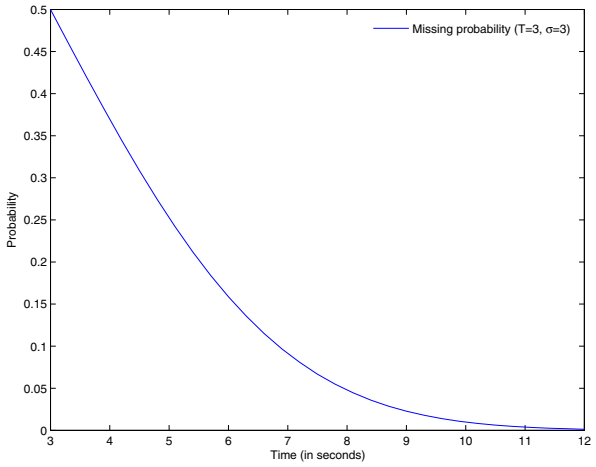

Figure 2. An example of the relationship between the waiting and the missing probability

and the variation of IDS delay is

$$Var(X) = E[(X-T)^2] = \sigma^2 \tag{6}$$

The cumulative distribution function will be

$$F(a) = \Phi\left(\frac{a-T}{\sigma}\right) \tag{7}$$

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-y^2/2} dy \tag{8}$$

is the cumulative distribution of a standard normal distribution (a normal distribution with parameter 1 and 0).

If we wait for time a and no intrusion is reported by the IDS, the probability that we mistakenly cut a branch in which the IDS may report an intrusion later is $1-F(a)$. We define $1-F(a)$ as the *missing probability*.

The missing probability is the probability that users will not be benefited from defensive execution since the useful backup execution has been mistakenly discarded. A branch cut by mistake will not affect the correctness of recovery in a self-healing system. The user has to suffer a recovery delay, which is normal in current self-healing systems.

Given the IDS delay is normally distributed with parameter $T = 3$ seconds and $\sigma^2 = 9$. Figure 2 shows the relationship between the waiting time and the missing probability.

If we wait for $4T = 12$ seconds and no intrusion has been reported, then the missing probability will be 0.0013 according to the above discussion.

We can select a proper k and wait kT for the IDS reports. The missing probability will be

$$1-F(a) = 1-\Phi\left(\frac{k-1}{\sigma}T\right) \tag{9}$$

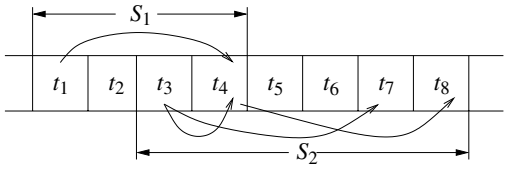

Figure 3. Overlap of Two Segments

We can take a smaller k for a risky branch cutting, which greatly reduces the cost of defensive execution but has the risk of cutting useful backup execution. We can also take a larger k for a conservative branch cutting with more cost and less risks. Our equations provide the missing probability guideline for selections with different strategies.

4.2 Time wrapping reports of the IDS

In a distributed system, message may arrive the destination not in the same order as they were sent. An IDS may identify damage not in the same order as the damage happened. These situations affect the progress of recovery analysis.

Definition 1 *If an IDS reports an incidents sequence* $i_1 i_2 \cdots i_n$, *where for any* i_j *and* i_k, $1 \le j < k \le n$, $i_j \prec i_k$ *then the IDS reports incidents* in the temporal order.

It is possible that IDSs do not report incidents according to the temporal order, then the situation in Figure 3 needs to be considered.

In the figure, curve lines are dependency relations. The IDS, or a message from other sites, firstly reports t_3 as a damaged transaction, which leads to t_4, t_7, and t_8 identified as damaged transactions. All these transactions are in segment S_2. Transaction t_1 may be reported as damaged after the incident reporting t_3. According to the dependency relations denoted by curve arrows in the figure, segment S_1 will be re-scanned. There will be a overlap between S_1 and S_2, where $S_1 \cap S_2 \neq \phi$.

The above situation is reflected as σ^2, the variation of the random variable in Equation 4. The detection of t_1 in Figure 3 simply has a longer delay than others. Therefore, whether the IDS reports intrusions in a temporally order does not affect our discussion on the missing probability.

5 Effectiveness

Defensive executions introduce larger throughput and more overload to the system. In this section we will discuss several arguments that affect the effectiveness of defensive executions.

5.1 Redundant rate

Given a transactional process defined by $G\langle V,E\rangle$, the actual executing path P, and m backup executing paths $P_1,P_2,\ldots,P_m$ in a specific defensive execution, we use the total number of executed transactions to measure the cost of the specific defensive execution.

We define

$$C_r = |(P_1 \cup P_2 \cup \ldots \cup P_m) - P| \tag{10}$$

as *redundant cost*. Please note that $P_i, 1 \leq i \leq m$ could be a partial path that does not cover the start node and the end node due to branch cutting. C_r is the general extra cost generated by a defensive execution.

Accordingly, we define

$$\alpha = \frac{C_r}{|P|} \tag{11}$$

as the *redundant rate*. A 100% redundant rate indicates that the number of backup transactions is as many as the number of transactions requested by the user. A 0% redundant rate indicates that there is no defensive execution at all.

In the worst case of defensive execution, C_r will be $|V - P|$, where we execute all transactions, or all paths, in the transactional processes as backups. In such situation, we have $P_1 \cup P_2 \cup \ldots \cup P_m = V$ (C_r could be even greater than $|V-P|$ if the situation defined in Theorem 1 happens, where a transaction may need more than one versions including backups.).

We will show that there exists an *effective upper bound* $\beta > 0$, such that for any defensive execution with $\alpha > \beta$, the situation will be worse than that without defensive executions. After finding out β, we can select an $0 < \alpha \leq \min(\frac{|V-P|}{|P|}, \beta)$ to make sure that the defensive execution is worth doing.

5.2 Measurements of effectiveness

Since we need to compare our techniques in this paper to that without defensive executions, we firstly did experiments on a prototype recovery system that was built based on our techniques developed in [15, 14] to create relationships between critical parameters.

All our experiments were done on a computer with a Pentium 4 2.4Ghz CPU and 512MB RAM. The database management system was PostSQL 7.4.5 under a Debian Linux with kernel version 2.6.8-2. The prototype system was developed with Java(TM) 2 Runtime Environment, Standard Edition, build 1.4.2_04-b04. To get stable results, each experiment lasts more than 300 seconds for given parameters, e.g., throughput.

The experimental results shown in Figure 4 demonstrates the relationship between the *turnaround time* (the duration time from the moment that a transaction enters the system to the moment that the transaction leaves the system) and the throughput (the number of transactions that the system processes in each second) of the system.

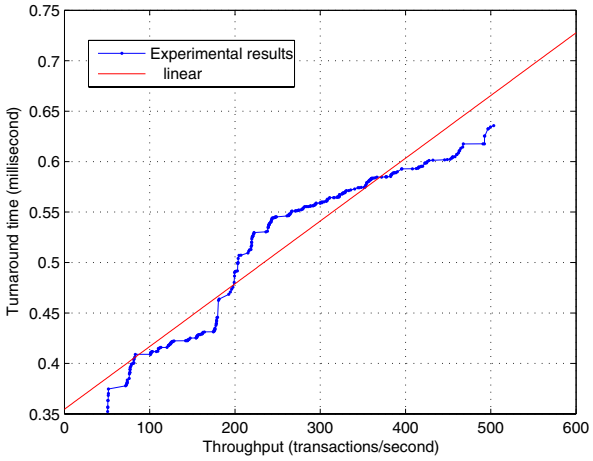

Figure 4. Turnaround time with a fitting function $y_1 = 0.00062191x + 0.35454$ and the norm of residuals $= 0.38164$

For simplicity, we can use a linear expression $f_1(x) = 0.00062191x + 0.35454$ to describe the relationship in Figure 4. Please note that we can also use other functions in stead of $f_1(x)$ for better accuracy. Function $f_1(x)$ can be determined by experimental study and curve fitting for any given system as we did.

We define $f_2(x) = xf_1(x)$ as the *system load function*. We can get the corresponding system load shown in Figure 5. For example, when the system has throughput 100 transactions/second, the turnaround time for each transaction is about 0.4 milliseconds. Therefore, 100 transactions can be done in 40 milliseconds. In each second, the system spends around $40/1000 = 4\%$ time on processing transactions. Hence the system load is 4% according to our definition.

Because most systems are running under unsaturated system load ($< 100\%$), in this paper, we are more interested in the linear part of the system load as shown in Figure 5.

The measurements of effectiveness are shown in Figure 6.

The top part of the figure shows an execution without defensive executions. After time T_e, if the execution was identified by the IDS as intruded, we need time T_r to recover the execution, which includes necessary damage assessment, undo tasks, and redo tasks [14, 15].

Figure 7 shows the experimental results of $\frac{T_r}{T_e}$, where we define *attacking density* as the number of malicious trans-

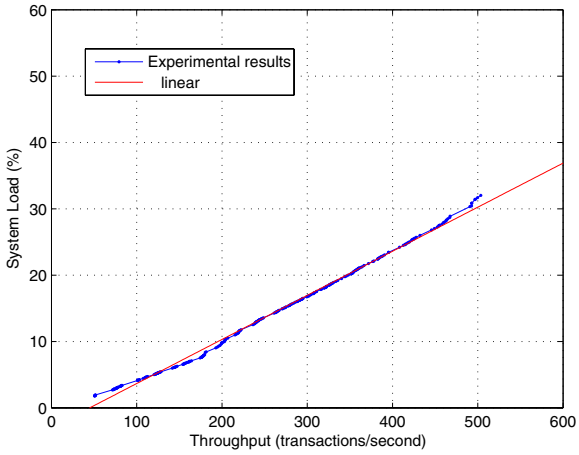

Figure 5. System load with a fitting function $y_2 = 0.066455x - 2.9864$ and the norm of residuals $= 9.3156$

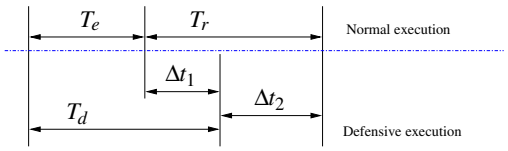

Figure 6. Time measurements of effectiveness

actions found in each second divided by the throughput.

According to our experimental results, the attacking density has more significant impact on $\frac{T_r}{T_e}$. With higher attacking densities, e.g., 24% and 34%, The recovery time T_r could be even longer than the original execution time T_e.

Assume that n transactions can be executed in time T_e without defensive executions. The bottom part of the figure shows the scenario of a defensive execution. To execute the same number of user's transactions as in T_e, a longer time $T_d > T_e$ needs to be spent, because the defensive execution has $(1+\alpha)n$ transactions to execute, where $\alpha > 0$. Since the throughput will be higher in defensive executions, the turnaround time of each transaction will increase accordingly. Thus the total execution time T_d will be longer.

In the figure, $\Delta t_1 = T_d - T_e$ is the increased total processing time caused by defensive execution. $\Delta t_2 = T_e + T_r - T_d$ is the reduced processing time gained by the defensive execution when attacks happen.

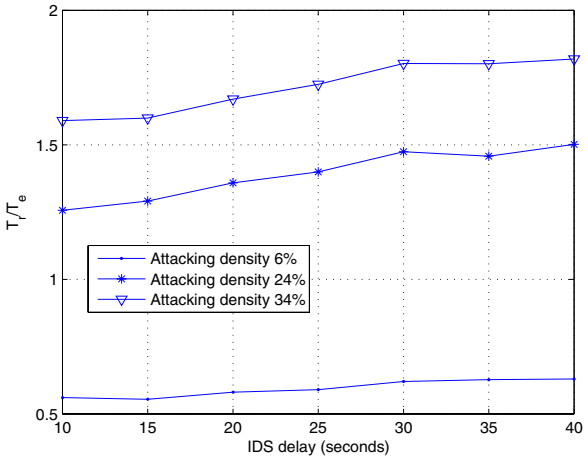

Figure 7. The experimental results of recovery time vs. executing time ($\frac{T_r}{T_e}$).

5.3 The effective upper bound of redundant rate—β

We classify applications into two types according to whether they are sensitive to the turnaround time of transactions.

5.3.1 In the case of turnaround time insensitive applications

The first type of applications is that the user's applications are not sensitive to the turnaround time while they are sensitive to the throughput of the transactional processing system. Such applications are processing systems for bank transactions, daily business transactions, etc. We will show that the value of Δt_1 can be ignored if we keep the system working on the linear part of system load.

Given a throughput u of user's transactions, the defensive execution will increase the throughput to $(1+\alpha)u$. As long as the system load is not saturated ($< 100\%$), user's application will not feel any performance degradation because the throughput of user's application is not changed. The user's throughput is only part of the whole throughput of the system.

If the highest throughput that the system can process is w, we have

$$(1+\alpha)u \leq w \tag{12}$$

to prevent the system load from being saturated. Therefore,

$$\alpha \leq \frac{w}{u} - 1 = \beta \tag{13}$$

where β is the upper bound of α to make the defensive execution effective. We call β as the *effective upper bound* of redundant rate.

In such situation, the defensive execution is transparent to the user's application. If attacks happen, the defensive execution can save as much time as T_r from recovery work. For example, in Figure 7, when the attacking density is 34% and the IDS delay is 15 seconds, T_r is about $1.6T_e$. In other words, if the IDS identifies an executing path as intruded after 15 seconds, we can save $1.6 \times 15 = 24$ seconds from the recovery.

The cost we paid for the defensive execution is the highest user's throughput we can achieve. Given α and w, the highest throughput u_{max} of user's application can be derived from Equation 12 as follows.

$$u_{max} \leq \frac{w}{1+\alpha} \tag{14}$$

The degradation of the highest throughput of user's application will be

$$w - u_{max} = \frac{\alpha w}{1+\alpha} \tag{15}$$

For example, as shown in Figure 4 and Figure 5, If normally the throughput of user's application is $u = 200$ transactions/second, then $\alpha = 1.5$ will not be a problem at all for the system. When $\alpha = 1.5$ the throughput of the whole system is $(1+\alpha)u = 500$ transactions/second. It is still in the linear part of the system load. α could be even larger since the system load is far from 100%.

Given a system with the capability of peak throughput 5000 transactions/second, if we select $\alpha = 4$, than the peak throughput of user's application will be reduced to 1000 transactions/second. Please note that the redundant cost is as four times many as the user's transactions. That is what we paid for saving the recovery time.

5.3.2 In the case of turnaround time sensitive applications

Another type of applications is that the user's applications are sensitive to the turnaround time of each transaction, or the total processing time of all transactions. Such applications are computing intensive applications, like decision supporting systems, etc. In such situation, the value of Δt_1 matters.

Assume that the throughput of users' transaction is C_1 without defensive executions. Given a redundant rate $\alpha > 0$, Δt_1 can be determined by the following equation.

$$\begin{aligned} \Delta t_1 &= C_1 T_e (f_1((1+\alpha)C_1) - f_1(C_1)) \\ &= C_1 T_e \int_{C_1}^{(1+\alpha)C_1} f_1'(x)dx \end{aligned} \tag{16}$$

where $f_1(x)$ is the turnaround time function of throughput, like the curve fitting function in Figure 4. Equation 16 explains the time cost we paid for defensive executions.

For a given IDS delay and attacking density, $r = \frac{T_r}{T_e}$ can be measured as shown in Figure 7. To let the defensive execution to be effective, we need to guarantee that

$$\Delta t_1 = C_1 T_e \int_{C_1}^{(1+\alpha)C_1} f_1'(x)dx \leq T_r = rT_e \tag{17}$$

Considering that the turnaround time function $f_1(x)$ is always incremental, thus

$$\alpha \leq \frac{1}{C_1} f_1^{-1}(\frac{r}{C_1} + f_1(C_1)) - 1 = \beta \tag{18}$$

With the above inequation, we can get the effective upper bound β.

For example, given $f_1(x) = 0.00062191x + 0.35454$, $C_1 = 100$, and $r = 1.6$ (when the IDS delay is 15 seconds and the attacking density is 34%), we can solve the inequation to get $\alpha \leq 1.49$. Thus $\beta = 1.49$ is the effective upper bound of redundant rate. Any larger redundant rate will cause that the defensive executions take longer time then the normal execution time plus recovery time.

Please note that we need to consider both types of applications for a real system. Assume that we get β_1 when we consider only the throughput as we did in Section 5.3.1, and we get β_2 when we consider the turnaround time, the final effective redundant rate should be $\min(\beta_1, \beta_2)$.

5.4 The accuracy of the IDS

No IDS system is perfectly accurate. False positive alarms (false alarms on legitimate activities) and false negative alarms (where the IDS failed to identify intrusions) can also affect the effectiveness of our techniques.

False positive alarms cause unnecessary branch cutting and switching of execution paths, which counteracts all benefits generated by defensive executions. However, the system incorporating our techniques just acts as a regular recovery system. It will not be worse. The users have to suffer the recovery time.

False negative alarms can only be solved by the intervention of administrators of the system. Since usually human intervention takes longer time than the IDS, the branch cutting will have been done. The user will not be benefited from our techniques.

In summary, in the worst case, an inaccurate IDS can force our techniques to work as a regular recovery system.

6 Related work

An Intrusion Detection System (IDS) [7] can detect intrusions. In a transactional processing system, the damages

directly caused by the attacker may be spread by executing legitimate transactions without being detected by the IDS. The IDS is unable to trace damage spreading and cannot locate all damage to the system.

The checkpoint [8, 9] techniques also do not work for efficient attack recovery. A checkpoint rolls back the whole transactional processes to a specific time. All work, including both damaged transactions and clean transactions after the specific time, will be lost, especially when the delay of the IDS is very long. In addition, checkpoints introduce extra storage cost.

The failure handling of transactional process has been discussed in recent work [4, 3, 12]. Failure handling occurs when the transactional process are in progress. When the IDS reports attacks, the malicious transactions usually have been successfully executed. Failure handling is not applicable because no failure occurred. Attack recovery is supposed to remove the effects of malicious transactions after they are committed..

Rollback recovery, e.g. [6, 2], is surveyed in [5]. It focuses on the relationship of message passing and considers temporal sequences based on message passing. In the message passing model, a process runs on a single site and errors occur at the end of the process. A transactional process usually runs on multiple sites and intrusions are detected at the middle of the process. Further more, rollback recovery handles errors while attack recovery handles intrusions. In brief, we are working on a different model for a different goal from theirs.

Previous work [1, 10, 14, 15] introduced different techniques to trace damage spreading and repair the damage in transactional processing systems. However, if attacks happen, all affected transactions will be rolled back (undone) and redone.

Rolling back and re-executing damaged transactions increase the response time of the system and may cause a significant processing delay. In such situations, the availability of the system is compromised and the system suffers the vulnerability of Denial of Service (DoS). Our technique can reduces the recovery delay with reasonable cost.

7 Conclusion

In this paper, we described a defensive executing technique against transactional level attacks. The defensive execution reduces the recovery delay by introducing extra executing cost if attacks happen. We also discussed the effectiveness of our technique and discussed conditions to make our technique to be more effective. Our work demonstrates that defensive executions are desirable if users of a self-healing transactional processing system prefer low processing latency under attacks.

Acknowledgment

We thank Professor Pierangela Samarati for her valuable and insightful comments. Our thanks also to the anonymous reviewers. Their comments were very helpful and greatly improved the quality of this paper. Peng Liu is partially supported by NSF CCR-TC-0233324.

References

[1] P. Ammann, S. Jajodia, and P. Liu. Recovery from malicious transactions. *IEEE Transaction on Knowledge and Data Engineering*, 14(5):1167–1185, 2002.

[2] Y. bing Lin and E. D. Lazowska. A study of time warp rollback machanisms. *ACM Transactions on Modeling and Computer Simulations*, 1(1):51–72, January 1991.

[3] Q. Chen and U. Dayal. Failure handling for transaction hierarchies. In A. Gray and P.-Å. Larson, editors, *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K*, pages 245–254. IEEE Computer Society, 1997.

[4] J. Eder and W. Liebhart. Workflow recovery. In *Conference on Cooperative Information Systems*, pages 124–134, 1996.

[5] E. N. M. Elnozahy, L. Alvisi, Y. min Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.

[6] D. R. Jefferson. Virtual time. *ACM Transaction on Programming Languages and Systems*, 7(3):404–425, July 1985.

[7] W. Lee and S. J. Stolfo. A framework for constructing features and models for intrusion detection systems. *ACM Transactions on Information and System Security*, 3(4):227–261, 2000.

[8] J.-L. Lin and M. H. Dunham. A survey of distributed database checkpointing. *Distributed and Parallel Databases*, 5(3):289–319, 1997.

[9] J.-L. Lin and M. H. Dunham. A low-cost checkpointing technique for distributed databases. *Distributed and Parallel Databases*, 10(3):241–268, 2001.

[10] P. Liu, S. Jajodia, and C. McCollum. Intrusion confinement by isolation in information systems. *Journal of Computer Security*, 8(4):243–279, 2000.

[11] R. A. Sahner, K. S. Trivedi, and A. Puliafito. *Performance and Reliability Analysis of Computer Systems*. Kluwer Academic Publishers, Norwell, Massachusetts, USA, 1996.

[12] J. Tang and S.-Y. Hwang. A scheme to specify and implement ad-hoc recovery in workflow systems. *Lecture Notes in Computer Science*, 1377:484–??, 1998.

[13] H. C. Tijms. *Stochastic Models*. Wiley series in probability and mathematical statistics. John Wiley & Son, New York, NY, USA, 1994.

[14] M. Yu, P. Liu, and W. Zang. Self-healing workflow systems under attacks. In *The 24th International Conference on Distributed Computing Systems(ICDCS'04)*, pages 418–425, 2004.

[15] M. Yu, P. Liu, and W. Zang. Multi-version based attack recovery of workflow. In *The 19th Annual Computer Security Applications Conference*, pages 142–151, Dec. 2003.