Multi-Version Attack Recovery for Workflow Systems

Meng Yu, Peng Liu, Wanyu Zang
School of Information Sciences and Technology, Pennsylvania State University, 16801
yumeng@psu.edu

Abstract

Workflow systems are popular in daily business processing. Since vulnerabilities cannot be totally removed from a system, recovery from successful attacks is unavoidable. In this paper, we focus on attacks that inject malicious tasks into workflow management systems. We introduce practical techniques for on-line attack recovery, which include rules for locating damage and rules for execution order. In our system, an independent Intrusion Detection System reports identified malicious tasks periodically. The recovery system detects all damage caused by the malicious tasks and automatically repairs the damage according to dependency relations. Without multiple versions of data objects, recovery tasks may be corrupted by executing normal tasks when we try to run damage analysis and normal tasks concurrently. This paper addresses the problem by introducing multi-version data objects to reduce unnecessary blocking of normal task execution and improve the performance of the whole system. We analyze the integrity level and performance of our system. The analytic results demonstrate guidelines for designing such kinds of systems.

1. Introduction

Increasingly, workflow management systems are being used as the primary technology for organizations to perform their daily business processes (workflows). A workflow consists of a set of tasks that are related to each other in terms of the semantics of a business process. Each task 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 workflow 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.

In this paper we mainly consider those intrusions that inject malicious tasks into the workflow management system instead of the attacks that only crash the workflow management system. These intrusions happen when attackers access a system with stolen (guessed, calculated, etc.) passwords or when some defense mechanisms, such as access control, are broken by the attackers. Under such intrusions, some tasks in a workflow may be forged or corrupted. Even worse, these malicious tasks will ultimately spread misleading information or damage to more tasks and nodes, generating more trash data in the workflow management system. The attack recovery on which we focus in this paper attempts to remove the malicious effects of the intrusions and to provide reasonable on-line recovery services. A motivating example for the attack recovery is illustrated in Figure 1.

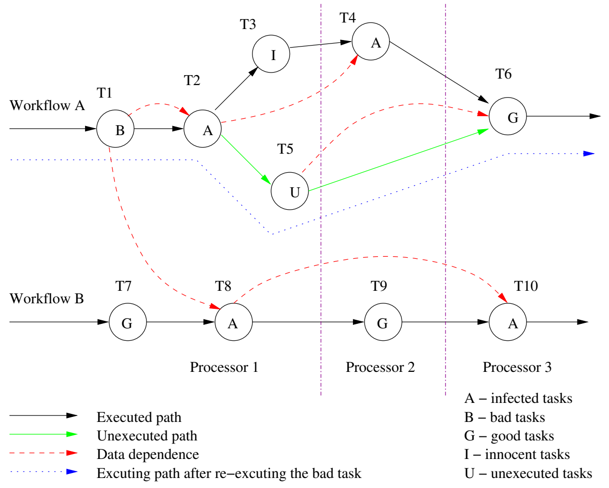


Figure 1. A Workflow

In the example, there are two workflows processed by three processors. Branches in the figure are not for parallel tasks but for selections of executing paths. $P_1 : T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_6$ and $P_2 : T_1 \rightarrow T_2 \rightarrow T_5 \rightarrow T_6$ are two different executing paths, which are selected by task T_2 in a specific execution. In this example, P_1 is the executing path led by attacks and P_2 is the normal executing path without attacks.

In the example, task T_1 marked with "B" is the only malicious task that is damaged directly by the attacker and is identified by the IDS. Due to reading dirty data from task T_1, task T_2, T_4, T_8 and T_{10} calculate wrong results. They are marked by "A", indicating infected tasks. Furthermore, task T_2, based on dirty data it reads from T_1, makes a wrong decision to execute tasks on path P_1. In fact task T_3 and task T_4 would not have been executed at all if T_1 were not damaged.

From this example, we learn that the IDS is unable to discover all damage to the system. The damage directly caused by the attacker will be spread by executing normal tasks without being detected by the IDS.

The example in Figure 1 shows how complex an attack recovery can be even for simple workflows with a single malicious task. We use it to explain our techniques in this paper. The example seems simple while it is complex enough to include all situations where a workflow can be affected by attacks. It is also powerful enough to demonstrate all aspects of our attack recovery theories.

Therefore, when attacks happen, we need to identify tasks that were affected and need to be undone. Then we need to identify tasks that need to be redone. In this paper, we will show that in some circumstances certain tasks that compute correctly need to be undone (e.g., task T_3 and task T_6) and some affected tasks may not need to be redone (e.g., task T_4), which is contrary, at least to some extent, to common knowledge on recovery. In addition, we need to execute recovery tasks and new workflow tasks in correct executing order in order to guarantee the correctness of attack recovery. Existing techniques do not effectively and efficiently solve the problem.

Within this paper, we introduce the fundamental theories for workflow attack recovery, which guarantee the correctness of recoveries. We propose an architecture and a scheduler algorithm for workflow attack recovery based on our theories. According to the set of malicious tasks reported by the IDS, our approach identifies all directly and indirectly infected tasks and repairs them with as little cost as possible. We break restrictions caused by dependency relations to achieve better performance. The performance and integrity level of our system are analyzed.

The rest of the paper is organized as follows. In Section 2 we introduce some definitions and notions used in this paper. Theories of multi-version data object based recovery are described and proved in Section 3. Our theories include the rules and conditions to find tasks affected by attacks and to guarantee the recovery correctness. We introduce an architecture and related algorithms in Section 4 based on our recovery theories. The performance and integrity level are analyzed in Section 5. We compare related work with ours in Section 6 and our conclusions are presented in Section 7.

2. Preliminaries

2.1. Set and Partial Orders

During the execution of workflow tasks and recovery tasks, there are some specific partial orders that need to be satisfied. In this section, we introduce some set and partial order relations related to our theories.

Preceding relations Given two tasks T_i and T_j, if task T_i is executed before T_j according to a workflow specification, or T_i occurs earlier than T_j in the system log, then T_i *precedes* T_j *by definition*, which is denoted by $T_i \prec T_j$.

Relation $\prec$ is a preceding relation defined by both workflow specification and system log. If T_i and T_j are tasks in the same workflow and they have not been executed, their relation is defined by the workflow specification. Otherwise, the relation is defined by their occurring sequence in the system log. If two tasks are within different workflows and have not been executed, then they have no defined $\prec$ relation.

In the example shown in Figure 1, a solid directed edge indicates a preceding relation. For example, $T_1 \prec T_2$, $T_2 \prec T_3$, $T_7 \prec T_8$ and $T_8 \prec T_9$. Relation $\prec$ is transitive. We can get $T_1 \prec T_3$ from $T_1 \prec T_2$ and $T_2 \prec T_3$. The relation $\prec$ is a partial order because some tasks have no preceding relations among them, such as T_4 and T_5 in the example.

When tasks of are executed in the workflow system, they have realistic preceding relations that are determined by the task scheduler. We use $T_i \prec_s T_j$ to denote that task T_i runs before T_j by scheduling, which we say task T_i *precedes* task T_j *by scheduling*.

Given any two tasks in the same workflow, if $T_i \prec T_j$ then T_i should be scheduled before T_j, namely $T_i \prec_s T_j$, and T_i will occur earlier than T_j in the system log. For two tasks within two different workflows and without a $\prec$ relation, or at least one of these two tasks is a recovery task, they will ultimately be scheduled by the scheduler and they will have $\prec_s$ relations. Before they are done and committed, they are neither defined by the same workflow specification nor in the system log. Therefore they have only $\prec_s$ relation, which is the difference between $\prec$ and $\prec_s$ relation.

Assuming that $\prec$ is a relation on set $\mathcal{S}$ then we define minimal$(\mathcal{S}, \prec) = x$ where $x \in \mathcal{S} \wedge \nexists x' \in \mathcal{S}, x' \prec x$. If $\mathcal{S}$ is a set including all tasks in Figure 1 then minimal$(\mathcal{S}, \prec) = T_1$. Note that there may be more than one result qualified to the definition of minimal$(\mathcal{S}, \prec)$. For example, if $\mathcal{S} = \{T_i, T_j, T_k\}$, $T_i \prec T_k$ and $T_j \prec T_k$, then both T_i and T_j are qualified results for minimal$(\mathcal{S}, \prec)$. In cases like these, we randomly select one qualified result as the value for minimal$(\mathcal{S}, \prec)$.

Data dependency and multi-version data objects We use $R(T)$ and $W(T)$ to denote the reading set and the writing set of task T. For example, given a task $T_x : x = a + b$, $R(T_x) = \{a, b\}$ and $W(T_x) = \{x\}$.

We introduce some concepts that are usually discussed in the field of parallel computing. Given two tasks $t_i \prec t_j$,

- If $(W(t_i) - \bigcup\limits_{t_i \prec t_k \prec t_j} W(t_k)) \cap R(t_j) \neq \phi$, then t_j is *flow dependent* on t_i, which is denoted by $t_i \rightarrow_f t_j$.
- If $R(t_i) \cap (W(t_j) - \bigcup\limits_{t_i \prec t_k \prec t_j} W(t_k)) \neq \phi$, then t_j is *anti-flow dependent* on t_i, which is denoted by $t_i \rightarrow_a t_j$.
- If $(W(t_i) - \bigcup\limits_{t_i \prec t_k \prec t_j} W(t_k)) \cap W(t_j) \neq \phi$, then t_j is *output dependent* on t_i, which is denoted by $t_i \rightarrow_o t_j$.

Intuitively, if $t_i \rightarrow_f t_j$, then t_j reads some data objects written by t_i. If $t_i \rightarrow_a t_j$, then t_j modifies some date objects after t_i reads them. If $t_i \rightarrow_o t_j$, then t_i and t_j have some common data objects that they modify.

Consider another task $T_b : b = x - 1$, where $T_x \prec T_b, R(T_b) = \{x\}$ and $W(T_b) = \{b\}$, we have $T_x \rightarrow_f T_b$ and $T_x \rightarrow_a T_b$. All the relations $\rightarrow_f, \rightarrow_a$ and $\rightarrow_o$ are data dependency relations and are not transitive. From the well known results of parallel computing, if a task T_j is data dependent on another task T_i then they cannot run in parallel and T_j should be executed after executing T_i, otherwise we will get wrong results.

With a single version of each data object, T_x must be executed before T_b to get the correct result. The order can be changed by introducing multiple versions of data objects. Suppose b^1 is one version of b with revision number 1, and b^2 is another version of b with revision number 2, and so on. The anti-flow dependency among T_x and T_b can be broken by revising $T_x : x = a + b^1$ and $T_b : b^2 = x - 1$. Even if T_b is executed before T_x, T_x still gets correct results by reading b^1. Multi-version data objects also can be used to break output dependencies.

By introducing multi-version data objects, restrictions caused by anti-flow and output dependencies are removed. Note that executing orders determined by flow dependencies are not changed by multiple-version data objects.

Control Dependency Given two tasks $T_i \prec T_j$ within the same workflow, if the execution of task T_j is decided by task T_i, then we say T_j is *control dependent* on T_i, which is denoted by $T_i \rightarrow_c T_j$. A control dependency relation is transitive. If $T_i \rightarrow_c T_j$ and $T_j \rightarrow_c T_k$ then $T_i \rightarrow_c T_k$. In the example shown in Figure 1, $T_2 \rightarrow_c T_3$, $T_2 \rightarrow_c T_4$ and $T_2 \rightarrow_c T_5$.

We use $\rightarrow$ to denote data or control dependency when the concrete type of dependency does not matter to our discussion. If there exist such tasks $T_1, T_2, \ldots, T_n, n \geq 2$ that $T_1 \rightarrow T_2 \rightarrow, \ldots, \rightarrow T_n$, then $T_1 \rightarrow^* T_n$.

If $T_i \rightarrow_c T_j$ then there are two possibilities about the execution of T_j: T_j should be executed or should not. We define two sets to describe these possibilities.

$\mathcal{S}_T(T_i)$ is a set of x, where $T_i \rightarrow_c^* x$ and x should be executed according to the executing result of T_i.

$\mathcal{S}_F(T_i)$ is a set of x, where $T_i \rightarrow_c^* x$ and x should not be executed according to the executing result of T_i.

$\mathcal{S}_T(T_i)$ is the set of task T_i's *true successors* and $\mathcal{S}_F(T_i)$ is the set of task T_i's *false successors*.

Please note the definition of these two sets are specific to concrete executions of workflows (or workflow instances). For the same task, these two sets may have different contents in different executions. If $T_j \in \mathcal{S}_T(T_i)$, then T_j is on the executing path according to the current execution. Otherwise T_j is not on the executing path according to the current execution. So, if there are more than two branches going out of task T_i within the workflow specification, only tasks on one branch belong to $\mathcal{S}_T(T_i)$ in a specific execution. Others belong to $\mathcal{S}_F(T_i)$. Consider the example in Figure 1, in the attacked execution $\mathcal{S}_T(T_2) = \{T_3, T_4\}$ and $\mathcal{S}_F(T_2) = \{T_5\}$. After we carry out the undo tasks and redo(T_2), it is another story. That is, $\mathcal{S}_T(\text{redo}(T_2)) = \{T_5\}$ should be in the recovered execution. Therefore, $\mathcal{S}_T(T_i)$ may be different from $\mathcal{S}_T(\text{redo}(T_i))$, which indicates that the recovered workflow may go through a different path from the previous path executed.

2.2. Workflows, recovery schemes and the system log

Based on the above definitions, workflows can be represented by $\langle \mathcal{T}, \prec, \rightarrow_c \rangle$, where $\mathcal{T}$ is the set of all workflow tasks, $\prec$ is the preceding relation on $\mathcal{T}$ and $\rightarrow_c$ is the control dependency relation on $\mathcal{T}$. Consider the example in Figure 1, $\mathcal{T} = \{T_i \mid 1 \leq i \leq 10\}$, $\prec = \{(T_1, T_2), (T_2, T_3), (T_3, T_4), (T_2, T_5), (T_4, T_6), (T_5, T_6), (T_7, T_8), (T_8, T_9), (T_9, T_{10})\}$ and $\rightarrow_c = \{(T_2, T_3), (T_2, T_4), (T_2, T_5)\}$. We do not limit our definition to a single workflow, here the set $\mathcal{T}$ may consist of tasks from more than one workflow.

Given workflows, data dependencies $\rightarrow_f, \rightarrow_a$ and $\rightarrow_o$ can be calculated from $\mathcal{T}$ while relations $\prec$ and $\rightarrow_c$ are defined only by workflows.

Similarly, we define a recovery scheme as $\langle \mathcal{R}, \prec_s, \rightarrow_c \rangle$, where $\mathcal{R}$ is the set of recovery tasks that consists of undo and redo tasks, $\prec_s$ is the scheduled preceding relation among the recovery tasks, and $\rightarrow_c$ is the control dependency relation among the recovery tasks.

Finally, given workflows $\langle \mathcal{T}, \prec, \rightarrow_c \rangle$, the system log is represented as $\langle \mathcal{L}, \prec \rangle$, where $\mathcal{L} \subseteq \mathcal{T}$ is the set of tasks that are completed and committed.

3. Theories of Multi-Version Data Object based Recovery

This section starts by introducing the concept of a revision history, which keeps all versions of data objects, then introduces rules for generating correct recovery schemes.

3.1. Revision History

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.

Each data object has a revision history with the 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 task 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 task 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*.

3.2. Operations on the Revision History

A normal task 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 tasks. It operates just as if multiple versions did not exist.

A recovery task, whether it is an undo or redo task, 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 tasks either. We can consider that recovery tasks 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 task 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 task. Any task that reads x^{v_k} needs to wait until the redo task 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 task is complete.

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

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

3.3. Axioms and Correctness Criteria

When attackers inject malicious tasks into the workflow management system the malicious tasks generate or corrupt some data objects directly. In addition, the data dependency relations and the control dependency relations among workflow tasks can further spread the damage to other data objects. We identify corrupted data objects, *dirty data objects*, based on the following two axioms.

Axiom 1 (Generated Dirty Data Objects) *Data objects generated by the tasks that should not have been executed are dirty.*

Axiom 2 (Spread Dirty Data Objects) *If a task computes using dirty data objects, the results are dirty.*

Concluded from the two axioms, Theorem 1 describes what kind of data should be cleaned within the workflow management system.

Theorem 1 *A data object is dirty if, and only if, it was generated in any of the following ways*

1. *Generated directly by a malicious task or corrupted directly by attackers*
2. *Calculated based on dirty data*
3. *Generated by a task that should not have been executed*
4. *Generated by a task that references data that is created by tasks that should have been executed, but did not*

Theorem 1 describes all possible patterns of damage spreading. In this paper we use the term *bad task* to represent a task that generates dirty data. Bad tasks consist of malicious tasks and affected tasks.

In Figure 1, task T_1 marked with 'B' was corrupted directly by attackers. So data that T_1 generates are dirty, which is indicated by item 1 in Theorem 1.

According to our definition of flow dependency, if T_j reads data that task T_i writes then $T_i \rightarrow_f T_j$. In Figure 1, task T_2 marked with 'A' is data dependent on task T_1. T_2 is affected by bad task T_1 because it reads dirty data from T_1 then creates wrong results which are also dirty. So does task T_4, which reads dirty data from T_2. T_8 and T_{10} fall in the same case, which is described by item 2.

The third situation described in item 3 is shown by task T_3 marked by 'I' in Figure 1, which is 'innocent'. In Figure 1 the execution of task T_3 is based on the executing result of task T_2. Since task T_2 is affected by T_1, it is possible that the selection of executing path is wrong. We need to redo task T_2 and then check whether T_3 is still a true successor of redo(T_2) in the recovered execution. If it is a false successor of redo(T_2) in the recovered execution, then the data T_3 generated before are dirty and T_3 needs to be undone, although the calculating results of T_3 are correct.

For the last case described in Theorem 1, please refer to the execution of task T_6 marked by 'G' in Figure 1. T_6 is flow dependent on task T_5 which was not executed in the attacked execution. When we redo task T_2, the workflow may be executed along a new path that continues with T_5. Then T_5 may generate different data from what T_6 has read in the attacked execution. Thus T_6 will get different results in the recovered execution. Therefore task T_6 produced a wrong result in the attacked execution and the data it generated are dirty.

The following definition describes the correctness criteria for our workflow attack recovery scheme.

Definition 1 *Given a set $\mathcal{N}$ of normal tasks, the recovery scheme $\langle \mathcal{R}, \prec_s, \rightarrow_c \rangle$ is correct if, and only if, the following conditions hold.*

1. *No dirty data exists after executing $\langle \mathcal{R}, \prec_s, \rightarrow_c \rangle$*
2. *No dirty data is generated by executing $\langle \mathcal{R}, \prec_s, \rightarrow_c \rangle$*
3. *The execution of normal tasks in $\mathcal{N}$ should not generate dirty data and should not have corrupted recovery tasks*
4. *The execution of tasks in $\mathcal{N}$ and $\mathcal{R}$ do not violate the definition of a workflow*

A correct recovery scheme is not isolated from the workflow management system. When we are carrying out the recovery there definitely exist some scheduled preceding relations between the recovery tasks and the normal workflow tasks. Condition 3 states that the execution of normal tasks should be clean. In other words, if a new task tries to read dirty data from some unrecovered tasks, it should be suspended for future execution until the data it tries to read are clean.

3.4. Generate Recovery Tasks

This section describes how to find undo and redo tasks. Since damage is spread by flow-dependencies and control dependencies, which are not affected by the revision history, we do not bother with different versions of a specific data object in this section.

From Theorem 1 and Definition 1 we can directly get the following theorem for undo tasks.

Theorem 2 (Undo tasks) *Assume $\mathcal{B}$ is a known set of bad tasks that need be undone. The correctness criteria will not be violated if, and only if, the following T_i and T_j are undone.*

1. $\forall T_i, T_i \in \mathcal{B}$
2. $\exists T_i \in \mathcal{B}, T_i \rightarrow_f^* T_j$
3. $\exists T_i \in \mathcal{B}, T_j \in \mathcal{L}, T_i \rightarrow_c^* T_j$*, and* $T_j \in \mathcal{S}_F(\mathit{redo}(T_i))$
4. $\exists T_i \in \mathcal{B}, \exists T_k \notin \mathcal{L}, T_i \rightarrow_c^* T_k, T_k \rightarrow_f^* T_j$*, and* $T_k \in \mathcal{S}_T(\mathit{redo}(T_i))$

PROOF SKETCH: The objective of undo tasks is to remove the dirty data in the workflow system. Each item in the theorem is exactly the formal description of the tasks that are described within the same numbered item in Theorem 1. □

We call the tasks described by condition 3 and condition 4 *candidate undo tasks* because we do not know if they really should be undone until redo(T_i) is executed.

Consider the problem in Figure 1 again. At the beginning, $\mathcal{B} = \{T_1\}$. Thus T_1 should be undone according to condition 1. T_2, T_4, T_8 and T_{10} should be undone because there exists $T_1 \in \mathcal{B}, T_1 \rightarrow_f T_2, T_2 \rightarrow_f T_4, T_1 \rightarrow_f T_8$ and $T_8 \rightarrow_f T_{10}$ (condition 2). At this point we know $\mathcal{B} = \{T_1, T_2, T_4, T_8, T_{10}\}$. T_3 should be undone because $\exists T_2 \in \mathcal{B}$, $T_3 \in \mathcal{L}$, $T_2 \rightarrow_c T_3$ and $T_3 \in \mathcal{S}_F(\text{redo}(T_2))$, which is specified by the condition 3. Now $\mathcal{B} = \{T_1, T_2, T_3, T_4, T_8, T_{10}\}$. Based on condition 4, task T_6 should be undone due to the fact that $T_2 \in \mathcal{B}$, $T_5 \notin \mathcal{L}$, $T_2 \rightarrow_c T_5, T_5 \rightarrow_f T_6$ and $T_5 \in \mathcal{S}_T(\text{redo}(T_2))$. Finally, we have $\mathcal{B} = \{T_1, T_2, T_3, T_4, T_6, T_8, T_{10}\}$.

The tasks that have already been undone and are still on the re-executing path should be redone to meet the specification of the workflow. Regarding redo tasks, we have the following theorem.

Theorem 3 (Redo tasks) *Assume $\mathcal{B}$ is a known set of bad tasks, $T_i \in \mathcal{B}$, then T_i should be redone if, and only if, any of the following conditions are satisfied.*

1. $\nexists T_j \in \mathcal{B}, T_j \rightarrow_c^* T_i$
2. $\exists T_j \in \mathcal{B}, T_j \rightarrow_c^* T_i, T_i \in \mathcal{S}_T(\mathit{redo}(T_j))$

PROOF SKETCH: In both cases, T_i has been damaged and is on the re-executing path. Thus T_i needs to be redone to meet the specification of the workflow. □

We call the tasks described by condition 2 *candidate redo tasks* because we do not know if they really should be redone until redo(T_j) is executed.

In Figure 1 task T_1, T_2, T_6, T_8 and T_{10} need to be undone. Since they are not control dependent on any bad task, they need to be redone, as stated in case 1 of Theorem 3. Since neither task T_3 nor task T_4 meets the requirements of Theorem 3, they do not need to be redone. They are simply not on the re-executing path of the workflow. Redoing them will generate dirty data because redoing them does not meet the specification of the workflow.

3.5. Partial Orders Caused by Dependency Relations

Since undo and redo tasks are not defined by workflow specifications we must create partial order relations among these tasks and normal workflow tasks to guarantee the correctness of recovery. The following two theorems give the partial order relations among recovery tasks and new workflow tasks.

As we mentioned before, in order to guarantee correct semantics, from the point of view of recovery tasks, there is only a single version for each object, so all data dependency relations are not broken among recovery tasks.

Theorem 4 (Partial ordering of recovery tasks) *Given the recovery tasks $\mathcal{R}$ and the system log $\langle \mathcal{L}, \prec \rangle$ the following rules derive scheduled precedence orders between any two tasks in $\mathcal{R}$.*

1. $T_i \prec T_j \Rightarrow \mathit{redo}(T_i) \prec_s \mathit{redo}(T_j)$
2. $T_i \rightarrow T_j \Rightarrow \mathit{redo}(T_i) \prec_s \mathit{redo}(T_j)$
3. $\forall T_i, \mathit{undo}(T_i) \prec_s \mathit{redo}(T_i)$
4. $T_i \rightarrow_a T_j \Rightarrow \mathit{undo}(T_j) \prec_s \mathit{redo}(T_i)$
5. $T_i \rightarrow_o T_j \Rightarrow \mathit{undo}(T_j) \prec_s \mathit{undo}(T_i)$
6. $T_i \rightarrow_c T_j, T_j \in \mathcal{S}_T(T_i) \Rightarrow \mathit{redo}(T_i) \rightarrow_c \mathit{redo}(T_j) \wedge \mathit{redo}(T_j) \in \mathcal{S}_T(\mathit{redo}(T_i))$
7. $T_i \rightarrow_c T_j, T_j \in \mathcal{S}_F(T_i) \Rightarrow \mathit{redo}(T_i) \rightarrow_c \mathit{redo}(T_j) \wedge \mathit{redo}(T_j) \in \mathcal{S}_F(\mathit{redo}(T_i))$
8. $T_i \in \mathcal{B}, T_j \in \mathcal{L}, T_i \rightarrow_c^* T_j$ *and* $T_j \in \mathcal{S}_F(\mathit{redo}(T_i))$ $\Rightarrow \mathit{redo}(T_i) \rightarrow_c \mathit{redo}(T_j) \wedge \mathit{undo}(T_j) \in \mathcal{S}_T(\mathit{redo}(T_i))$
9. $T_i \in \mathcal{B}, \exists T_k \notin \mathcal{L}, T_i \rightarrow_c^* T_k, T_k \rightarrow_f^* T_j$ *and* $T_k \in \mathcal{S}_T(\mathit{redo}(T_i))$ $\Rightarrow \mathit{redo}(T_i) \rightarrow_c \mathit{undo}(T_j) \wedge \mathit{undo}(T_j) \in \mathcal{S}_T(\mathit{redo}(T_i))$
10. $T_i \in \mathcal{B}, \exists T_j \in \mathcal{B}, T_j \rightarrow_c^* T_i, T_i \in \mathcal{S}_T(\mathit{redo}(T_j)) \Rightarrow \mathit{redo}(T_i) \rightarrow_c \mathit{redo}(T_j) \wedge \mathit{redo}(T_j) \in \mathcal{S}_T(\mathit{redo}(T_i))$

PROOF: See appendix.

Usually we prefer not to stop the services of the workflow system while carrying out the recovery. For the purpose of running both the recovery tasks and normal workflow tasks concurrently, let us see what will happen when using a single copy of data objects, which means an old value will be lost after writing.

We have the following theorem using the assumption of single-version data objects.

Theorem 5 (Partial ordering of normal tasks) *Given the recovery tasks $\mathcal{R}$, new workflow tasks $\mathcal{N}$, and the system log $\langle \mathcal{L}, \prec \rangle$, if every data object has only one copy, the following rules derive scheduled preceding orders for any two tasks in $\mathcal{R} \cup \mathcal{N}$.*

1. $T_i \prec T_j, T_i, T_j \in \mathcal{N} \Rightarrow T_i \prec_s T_j$
2. $(T_i \rightarrow_f T_j) \vee (T_i \rightarrow_a T_j) \vee (T_i \rightarrow_o T_j) \vee (T_i \rightarrow_c T_j), T_j \in \mathcal{N} \Rightarrow \mathit{undo}(T_i) \prec_s \mathit{redo}(T_i) \prec_s T_j$
3. $T_i \rightarrow_c^* T_k, T_k \rightarrow_f^* T_j, T_k \notin \mathcal{L} \cup \mathcal{N}, T_j \in \mathcal{N} \Rightarrow \mathit{undo}(T_i) \prec_s \mathit{redo}(T_i) \prec_s T_j$

PROOF: See appendix.

Theorem 5 shows that normal tasks cannot be executed before the generation of a recovery scheme is complete. In other words, we can run normal tasks if, and only if, all malicious tasks reported by the IDS have been processed, which may cause temporary delay of services when attacking rate is high.

Theorem 5 is derived from the assumption that every data object has one copy. If the data object was changed the value would be lost. Theorem 5 tells us that achieving no stop service cannot without multiple copies (versions) of data objects and full information about possible normal tasks and recovery tasks. Multi-version data objects can solve the problem.

In Theorem 5, rule 1 states that the execution of normal tasks should abide by the specification of workflows. Rule 2 and rule 3 describe restrictions of executing orders among recovery tasks and normal tasks. With the use of revision histories, anti-flow and output dependencies are broken while flow and control dependencies are not. From the proof of Theorem 5 we know that running recovery tasks and normal tasks in parallel with their corresponding revision history will not corrupt recovery tasks while we must take the risks of corrupting normal tasks. Since we can still chase damage spreading by dependency relations, corruption of normal tasks is not a problem and damage can be repaired later.

4. The Recovery System

4.1. Revision History

A revision history can be built at the system level or application level. In fact, it does exist in the system log or application level. For example, some system logs record different versions of data objects written at different times. A history of a bank account is also a revision history.

Besides the revision history, the system needs to record which versions a specific task reads and which versions the task writes, recording this information is also implemented in modern workflow systems.

4.2. Architecture

The architecture of an attack recovery system for workflows is shown in Figure 2.

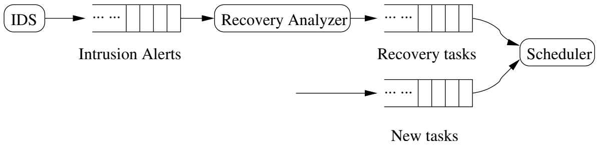


Figure 2. Processing Structure of an Attack Recovery System

In this architecture, there are two independent processing parts, the recovery analyzer and the task scheduler. An independent Intrusion Detection System (IDS) reports malicious tasks to the system by periodically putting IDS alerts in a queue. The recovery analyzer processes IDS alerts one by one. It determines the amount of damage and generates recovery tasks. The recovery tasks are sorted and put into another queue. The tasks scheduler schedules recovery tasks and normal tasks concurrently.

4.3. Algorithms

The recovery analyzer runs the following algorithm.

1. Wait until the queue of IDS alerts is not empty. Get one IDS alert and continue.
2. Determine all damage caused by the attack that the IDS reported and generate undo tasks according to Theorem 2. Abort and block all running tasks that are dependent on damaged tasks.
3. Generate redo tasks according to Theorem 3. For all $T_i : undo(T_i) \in \mathcal{R}$, if $redo(T_i) \notin \mathcal{R}$, mark $undo(T_i)$ as $undo(T_i)_b$. Otherwise, mark $redo(T_i)$ as $redo(T_i)_b$.
4. Set up precedence orders among recovery tasks according to Theorem 4.
5. Sort recovery tasks and put them into the queue of recovery tasks.
6. Goto step 1.

There is no special order that needs to be satisfied while scheduling normal and recovery tasks. After a recovery task T_b is done, release tasks that are blocking on it and provide them data objects with revision number t_m if the number exists, otherwise with revision number t_{m-1}.

5. Evaluation

5.1. Impacts on the System by the IDS

Our techniques do not depend on timely reporting by the IDS. As long as the damage is reported, whether it is reported by the IDS or the administrator of a system, our techniques work out all affected tasks and repair them. However, if the reporting delay is significant, more tasks in the system will be affected before recovery, which leads to more time being spent on recovery. Any system that has an IDS suffers with this problem, except intrusion masking systems [15].

Since we cannot guarantee that an IDS is 100% accurate, we have to depend on the administrator of a system to compensate for the inaccuracy. The administrator may revise the damaged-task set $\mathcal{B}$ according to further investigation. Consequently, the recovery system repairs the damage newly reported by the administrator. Since our techniques do not depend on timely reporting, the delay from the administrator is acceptable.

5.2. Performance

There are two queues in the system. The analyzer and the scheduler work independently if both queues are not full. Suppose the arrival of IDS alerts has a Poisson distribution with rate λ_1, the time distribution of processing IDS alerts is exponentially distributed with parameter μ_1, and the time distribution of executing recovery tasks is also exponentially distributed with parameter μ_2. The system behaves like a tandem Jackson network.

We are interested in relationships among λ_1, μ_1 and μ_2 while considering the loss probability of IDS alerts.

While $\mu_2 \leq \mu 1$, the queue of recovery tasks becomes the bottleneck of the system. The queue of recovery tasks will be full and no further recovery tasks could be generated. Therefore, no further IDS alerts could be processed. This situation should be avoided.

While $\mu_1 < \mu_2$, the queue of IDS alerts becomes the bottleneck of the system. The loss probability of IDS alerts is determined by the processing rate μ_1. Consider the buffer size of the queue is K. According to our assumptions the queue becomes a $M/M/1/K$-queue [14, 12]. The steady state probability for such a queue is given by:

$$p_0 = \frac{1-\rho}{1-\rho^{K+1}} \quad (1)$$

$$p_k = p_0 \rho^k \quad (2)$$

where $1 \leq k \leq K$, $\rho = \frac{\lambda_1}{\mu_1}$, and p_k indicates that there are k IDS alerts in the queue. Let us assume $\lambda_1 < \mu_1 < \mu_2$, $\rho < 1$. We have

$$E[N] = \sum_{k=0}^{K} kp_k = \frac{\rho}{1-\rho} - \frac{K+1}{1-\rho^{K+1}}\rho^{K+1} \qquad (3)$$

Thus,

$$E[T] = \frac{E[N]}{\lambda_1} = \frac{1}{\mu_1 - \lambda_1} - \frac{K+1}{\mu_1 - \lambda_1 \rho^k}\rho^k \qquad (4)$$

5.3. Integrity Level

Suppose the expected delay time of IDS reports is $E[T']$, the total time an IDS alert exists in the system is $E[T'] + E[T]$. During that time, $l = \mu_3(E[T'] + E[T])$ tasks have been executed, where μ_3 is the executing rate of normal tasks. In fact, l is the number of tasks that the analyzer is supposed to scan in the system log. The larger l is, the more unidentified bad tasks there are, and the longer time the analyzer takes to scan the log.

l describes the integrity level of the system. In other words, how many tasks exist where it is unknown whether they are infected or not. Although reducing μ_3 improves integrity, it will enlarge the degradation of performance simultaneously. An extreme situation is where $\mu_3 = 0$, in other words, stop service to normal tasks. Then no further damage occurs. We can also reduce the delay time of IDS reports and increase the processing speed of IDS alerts to improve the integrity level of the system without sacrificing the performance μ_3, which is what we usually do.

Another parameter affecting the integrity level of the system is *loss probability* of IDS alerts. The loss probability is given by:

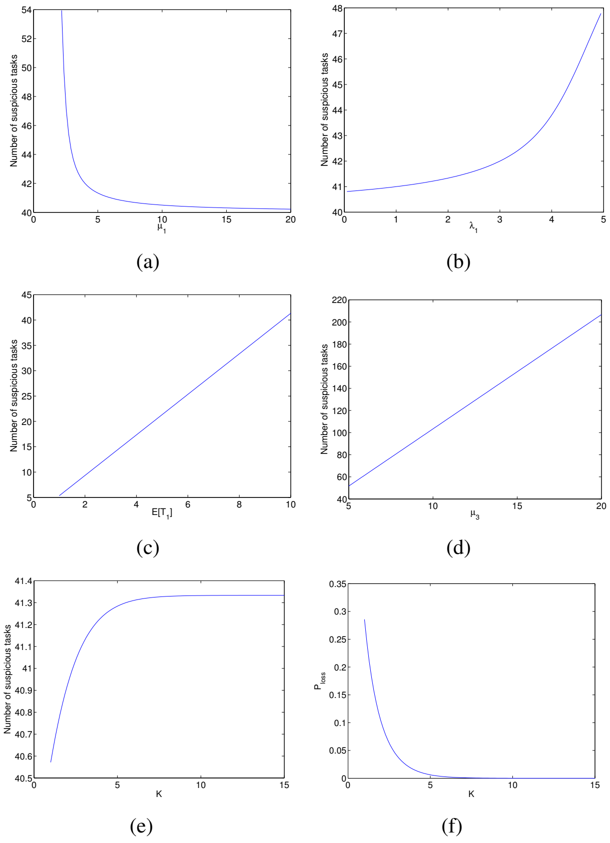
$$p_{loss} = p_K = \frac{\rho^K - \rho^{K+1}}{1 - \rho^{K+1}} \qquad (5)$$

The higher p_{loss} is, the more unidentified malicious tasks there are in the system.

The queue of recovery tasks is the second queue of the tandem Jackson network. We assume that the probability of its overflow [8] is relative small compared with the loss probability of the first queue since $\mu_1 < \mu_2$.

5.4. A Case Study

A case study is more intuitive than equations. We set up a case, where $E[T'] = 10, K = 20, \mu_1 = 5, \lambda_1 = 2, \mu_3 = 4$, to investigate the integrity level of the system. When parameters change, the results are shown in Figure 3.



(a)

(b)

(c)

(d)

(e)

(f)

Figure 3. Impacts on the system with different parameters

In this case, K has little effect on l after it is large enough, e.g., > 5 in the figure. Both $E[T']$ and μ_3 have linear impacts on l while μ_3 has relatively more significant impact on l. As for μ_1 and λ_1, as soon as $\rho = \frac{\lambda_1}{\mu_1}$ is small enough, their changes have little effect on l.

6. Related Work

An Intrusion Detection System (IDS) [9] can detect some intrusions. But, in a workflow system, the damages directly caused by the attacker may be spread by executing normal tasks without being detected by the IDS. The IDS is unable to trace damage spreading and cannot locate all damage to the system.

The checkpoint [10, 11] techniques also do not work for efficient workflow recovery. A checkpoint rolls back the whole workflow system to a specific time. All work, including both malicious tasks and normal tasks 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 work most similar to ours handles malicious transactions in a database system, as discussed in [1]. When intrusions have been detected by the IDS, the database system isolates and confines the impaired data. Then, the system carries out recovery for malicious transactions. This work is different from ours in that they consider little about relations among transactions; the work is unable to trace damage spreading and cannot locate all damage to the system. In contrast, we show that to guarantee the correct recoveries of a workflow, we need all data and control dependency relations among transactions. Otherwise, both recovered and newly executed transactions could be corrupted.

The failure handling of workflow has been discussed in recent work [5, 4, 13]. Failure handling is different from attack recovery in two aspects. On one hand, the two areas have different goals. Failure handling tries to guarantee the atomicity of workflows. When failure happens, their work finds which tasks should be aborted. If all tasks are successfully executed, failure handling does nothing for the workflow. Attack recovery has different goals, which need to do nothing for failure tasks even if they are malicious, because malicious failure tasks have no effect on the workflow system. Attack recovery focuses on malicious tasks that are successfully executed. It tries to remove all effects of such tasks. On the other hand, these two systems are active at different times. Failure handling occurs when the workflows are in progress. When the IDS reports attacks, the malicious tasks usually have been successfully executed. Failure handling is not applicable because no failure occurred. Attack recovery is supposed to remove the effects of malicious tasks after they are committed..

Rollback recovery, e.g. [7, 3], is surveyed in [6]. It focuses on the relationship of message passing and considers temporal sequences based on message passing. In contrast to their research, we focus on data and control dependency relations inside workflow tasks. In fact, message passing is a kind of data dependency relation but not vice versa (e.g., a data dependency relation caused by more than one message passing step or by sharing data). We also observed that in workflow recovery an execution path may change due to control dependencies, causing different patterns of message passing. In addition, our methods exploit more detail in dependency relations than the methods that are message-passing based; therefore our method is more effective and efficient for workflow recovery.

Decentralized workflow processing is becoming more and more popular. In distributed workflow models, workflow specifications cannot be accessed at a central node. They are carried by the workflow itself or stored in a distributed manner. In either case, our theories are still practical. We need to process the specifications of workflows in a distributed manner.

In some work such as [2], security and privacy are important, and the whole specification of workflows avoids being exposed to all processing nodes to protect privacy. Our theories are based on the dependency relations among tasks. The specification can be best protected by exposing only dependency relations to the recovery system.

7. Conclusions and Future Work

We described fundamental theories for on-line attack recovery of workflows. While an independent IDS reports malicious tasks periodically, our techniques find all damage caused by the malicious tasks and repair them automatically. We introduced restrictions of executing order that exist in an attack recovery system. We partially removed the restrictions by introducing multi-version data objects to reduce unnecessary blocks in order to reduce degradation of performance while carrying out the recovery. We evaluated the performance and integrity level of such systems. We will compare our multi-version data-objects-based systems with single-version data-objects-based systems in the near future.

Acknowledgment

We thank LouAnna Notargiacom for her valuable and insightful comments. She also polished this paper to improve the presentation. Our thanks also to the anonymous reviewers. This work was supported in part by DARPA and AFRL, AFMC, USAF, under award number F20602-02-1-0216, by NSF CCR-TC-0233324, and by Department of Energy Early Career PI Award.

References

[1] P. Ammann, S. Jajodia, and P. Liu. Recovery from malicious transactions. *IEEE Transaction on Knowledge and Data Engineering*, 2002.

[2] V. Atluri, S. A. Chun, and P. Mazzoleni. A chinese wall security model for decentralized workflow systems. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 48–57. ACM Press, 2001.

[3] 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.

[4] 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.

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

[6] 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.

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

[8] D. P. Kroese and V. F. Nicola. Efficient simulation of a tandem jackson network. *ACM Transactions on Modeling and Computer Simulation*, 12(2):119–141, April 2002.

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

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

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

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

[13] 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.

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

[15] M. Yu, P. Liu, and W. Zang. Intrusion masking for distributed atomic operations. In *The 18th IFIP International Information Security Conference*, Athens Chamber of Commerce and Industry, Greece, 26-28 May 2003. IFIP Technical Committee 11, Kluwer Academic Publishers.

Appendix: Proof of theorems

Proof of Theorem 4

1. Comes directly from criterion 4 of Definition 1.
2. Derived from rule 1 and the definition of relation $\rightarrow$
3. By contradiction. If redo$(T_i) \prec_s$ undo(T_i) then the effects of task T_i will be undone, which violates criterion 4 of Definition 1.
4. By contradiction. Assume redo$(T_i) \prec_s$ undo(T_j). Since $T_i \rightarrow_a T_j, R(\text{redo}(T_i)) = R(T_i)$ and $W(\text{undo}(T_j)) = W(T_j)$, so $R(\text{redo}(T_i)) \cap W(\text{undo}(T_j)) \neq \phi$. Moreover, $W(\text{undo}(T_j))$ is dirty before undo(T_j). Therefore redo(T_i) reads dirty data from $R(\text{redo}(T_i)) \cap W(\text{undo}(T_j))$ then generates dirty data, which violates criterion 2 of Definition 1.
5. By contradiction. From $T_i \rightarrow_o T_j$ we have $T_i \prec T_j$ and $W(T_i) \cap W(T_j) \neq \phi$. Then in the system log $W(T_i)$ has an older version than $W(T_j)$ for $W(T_i) \cap W(T_j)$. If undo$(T_i) \prec_s$ undo(T_j) then $W(T_i) \cap W(T_j)$ was not undone for T_i. In other words, T_i was not undone completely, which violates criterion 1 of Definition 1.
6. Comes directly from criterion 4 of Definition 1.
7. Comes directly from criterion 4 of Definition 1.
8. Comes directly from the condition 3 of Theorem 2.
9. Comes directly from the condition 4 of Theorem 2.
10. Comes directly from the condition 2 of Theorem 3. □

Proof of Theorem 5

1. Comes directly from criterion 4 of Definition 1.
2. undo$(T_i) \prec_s$ redo(T_i) comes directly from the rule 3 of Theorem 4.

   If T_j is data dependent on T_i we prove the result by contradiction. Since $R(\text{redo}(T_i)) = R(T_i)$ and $W(\text{redo}(T_i)) = W(T_i)$ so if $T_i \rightarrow T_j$ then redo$(T_i) \rightarrow T_j$. There are three cases.

   - redo$(T_i) \rightarrow_f T_j$. When T_j reads data from $W(\text{redo}(T_i)) \cap R(T_j)$ the redo(T_i) has not created it. So the task T_j gets wrong data and is corrupted.
   - redo$(T_i) \rightarrow_o T_j$. After executing T_j, redo(T_i) writes $W(\text{redo}(T_i)) \cap W(T_j)$ again. So the executing results of redo(T_i) in $W(\text{redo}(T_i)) \cap W(T_j)$ is lost. Therefore the task redo(T_i) is corrupted.
   - redo$(T_i) \rightarrow_a T_j$. redo(T_i) will read data that T_j writes in $R(\text{redo}(T_i)) \cap W(T_j)$. But according to the definition of workflow, redo(T_i) should read data that exists in $R(\text{redo}(T_i)) \cap W(T_j)$ before executing T_j. So the task redo(T_i) is corrupted.

   In these cases, either the new task T_j is corrupted, which violates criterion 3 of Definition 1, or the recovery task redo(T_i) is corrupted, which violates criterion 2 of Definition 1.

   If T_j is control dependent on T_j then the execution of T_j depends on the executing result of T_i. If $T_j \prec_s$ redo(T_i) then it is possible that $T_j \in \mathcal{S}_F(\text{redo}(T_i))$ after redo(T_i) is done. In this case, T_j creates dirty data according to Theorem 1 therefore the execution of T_j violates both criterion 3 and criterion 4 of Definition 1.
3. undo$(T_i) \prec_s$ redo(T_i) comes directly from rule 3 of Theorem 4.

   We prove redo$(T_i) \prec_s T_j$ by contradiction. Since redo$(T_i) \in \mathcal{R}$ is not done, we do not know: if $T_k \in \mathcal{S}_T(\text{redo}(T_j))$. Assume $T_j \prec_s$ redo(T_i). After the redo(T_i) is done, it is possible that $T_k \in \mathcal{S}_T(\text{redo}(T_j))$. According to the condition 4 of Theorem 2, T_j should be undone because it creates dirty data, which violates criterion 3 of Definition 1. □