Self-Healing Workflow Systems under Attacks

Meng Yu, Peng Liu and Wanyu Zang
School of Information Sciences and Technology, Pennsylvania State University

***Abstract*— Workflow systems are popular in daily business processing. Since vulnerability cannot be totally removed from a workflow management system, successful attacks always happen and may inject malicious tasks or incorrect data into the workflow system. Referring to the incorrect data will further corrupt more data objects in the system, which comprises the integrity level of the system. This problem cannot be efficiently solved by existing defense mechanisms, such as access control, intrusion detection, and checkpoints. In this paper, we propose a practical solution for on-line attack recovery of workflows. The recovery system discovers all damages caused by the malicious tasks that the intrusion detection system reports and automatically repairs the damages based on data and control dependencies among workflow tasks. We analyze the behaviors of our attack recovery system based on the Continuous Time Markov Chain model. Finally, we address how to design a practical recovery system step by step based on the analytical results.**

I. Introduction

Increasingly, workflow management systems become 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 focus on those intrusions that inject malicious tasks into the workflow management system instead of the attacks that only crash the workflow management system. These intrusions can happen in many situations, for example, 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, tasks and data in a workflow may be forged or corrupted. For one example, an attacker may forge bank transactions to steal money from accounts of others, thereby generating malicious workflow tasks. For another example, the attacker may schedule a travel with forged credit card information that carries incorrect data in workflow tasks.

Even worse, these malicious tasks will ultimately spread misleading information or damage to more tasks and processing nodes, generating more trash data in the workflow management system. To correct the situation, the malicious tasks must be removed from the workflow system, and all affected tasks must be repaired.

A motivating example for workflow attack recovery is illustrated in Figure 1. In the example, two workflows are processed by three processors. Branches in the figure are not for parallel tasks, but rather they are for choices of execution paths. $P_1 : t_1t_2t_3t_4t_6$ and $P_2 : t_1t_2t_5t_6$ are two execution paths, but in each execution (or instance), only one path can be selected by t_2. In this example, P_1 is the execution path led by an attack, and P_2 is the normal execution path without corruption.

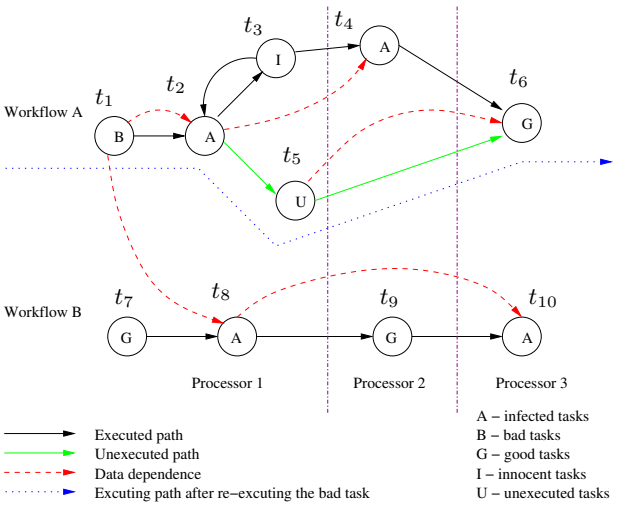


Fig. 1. An example of workflows

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 Intrusion Detection System (IDS) [1]. Due to reading corrupted data from task t_1, tasks t_2, t_4, t_8, and t_{10} calculate wrong results. They are marked with "A", indicating infected tasks. Furthermore, task t_2, based on the corrupted 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 trace damage spreading and cannot locate all damages to the system. The damages directly caused by the attacker may be spread by the execution of normal, legitimate tasks without being detected by the IDS.

The checkpoint [2] techniques do not work either for efficient workflow attack 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.

When attacks happen, we need to identify the tasks that

were affected and need to be undone. Then we need to identify the tasks that should be redone. In this paper, we will show that in some circumstances certain tasks that compute correctly may need to be undone (e.g., task t_3 and task t_6) while 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. Finally, we need to execute recovery tasks and new workflow tasks in correct order. The example in Figure 1 shows how complex an attack recovery can be even for simple workflows with a single malicious task.

Existing techniques cannot effectively and efficiently solve the problem. In this paper, we propose a practical solution for on-line attack recovery of workflows. Our main contributions are as follows. First, we present the fundamental theories for workflow attack recovery. Given the set of malicious tasks reported by the IDS, our approach is able to identify all directly or indirectly damaged tasks and repair them on-line. Second, we propose a software architecture to implement the attack recovery system. Third, we build a Continuous Time Markov Chain model to evaluate the performance of the proposed attack recovery system. The analytical results indicate that our system is practical. Finally, we introduce guidelines for designing a effective recovery system.

The rest of the paper is organized as follows. In Section II, we introduce some definitions and notions used in this paper. Our theories of attack recovery are described in Section III, including the rules to find affected tasks and the rules to determine execution orders of tasks. We present the attack recovery system architecture in Section IV based on our recovery theories. The behaviors of the system are analyzed in Section V. We summarize our analysis and address how to design a practical attack recovery system in Section VI. We compare related work with ours in Section VII and conclude our work in Section VIII.

II. Preliminaries

A. Execution paths, the system log, and traces

A workflow can be specified by a directed graph $\langle V, E\rangle$, where V is the set of vortex (tasks), and E is the set of directed edges (immediate precedence relations). If $(t_i, t_j) \in E$, then t_j should be executed *subsequently* to t_i.

A workflow $G(V, E)$ has a *start node* with 0-indegree, and some *end nodes* with 0-outdegree. Any path from the start node to the end node is an *execution path*. Please note that a task may be repeated in a execution path because there may be circles in the workflow. Different visits to the same node in a path are different instances of the task, which are distinguished by superscripts, such as t_i^1, t_i^2, and so on. For example, $t_1t_2t_3t_4t_6$, $t_1t_2t_5t_6$, and $t_1t_2^1t_3^1t_2^2t_3^2t_4t_6$ are three executing paths in Figure 1 .

In the workflow system, a task does not take effect until it is committed to the system. We assume that the committing time is distinguishable. The *system log* is a sequence of tasks $t_1, t_2, \ldots, t_n$, where $t_i, 1 \leq i < n$ is committed earlier than t_{i+1}. We denote the system log by $\mathcal{L}$. The system log in Figure 1 could be $\mathcal{L}_1 = t_1t_7t_2t_8t_3t_4t_9t_6t_{10}$.[1]

The *trace* of a workflow is the subsequence of the system log consisting of tasks in the workflow. In the trace of a workflow, only one execution path is completed. Given a trace $t_1t_2 \ldots t_it_{i+1} \ldots t_n$, we define $succ(t_i)$, t_i's *successors* set, as $\{t_k \mid i+1 \leq k \leq n\}$. For example, $succ(t_2)$ is $\{t_3, t_4, t_6\}$, which also indicates that another execution path $t_1t_2t_5t_6$ was not selected in the instance of the workflow.

B. Precedence relations

In the system log, if t_i appears earlier than t_j, t_i is a *predecessor* of t_j, which is denoted by $t_i \prec t_j$. Note that two tasks in two different workflows may have a precedence relation defined by the system log. For example, $t_1 \prec t_8$ in $\mathcal{L}_1$.

Although the precedence is defined by the system log, given two tasks in the same workflow, we can know their possible precedence relation *before* executing them: for example, $t_1 \prec t_2$, $t_2 \prec t_4$, $t_2 \prec t_5$, and $t_2^i \prec t_2^j$, where $i < j$, and so on.

Please note that $t_2 \prec t_4$ and $t_2 \prec t_5$ cannot hold simultaneously since in one execution of the workflow, only one execution path is chosen.

Relation $\prec$ is transitive and asymmetric. 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 precedence relations among them, such as t_4 and t_5 in the example.

Assume $\prec$ 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 there may be more than one result qualified by the definition of minimal$(\mathcal{S}, \prec)$. For example, $\mathcal{S} = \{t_i, t_j, t_k\}$, $t_i \prec t_k$ 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 of minimal$(\mathcal{S}, \prec)$. The task scheduler is supposed to choose the minimal$(\mathcal{S}, \prec)$ to execute.

C. Data dependency

We use $R(T)$ and $W(T)$ to denote the reading set and the writing set of task T. For example, given two tasks $t_x : x = a + b$, and $t_b : b = x - 1$. $R(t_x) = \{a, b\}$, $W(t_x) = \{x\}$, $R(t_b) = \{x\}$, and $W(t_b) = \{b\}$.

We introduce some concepts that are usually discussed in the field of parallel computing.

Definition 1: Given two tasks $t_i \prec t_j$,

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

[1] Since the workflow could be processed in a distributed style, the system log may be stored in segments. But it does not affect our discussion.

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 to modify. Consider tasks t_x and t_b, where $t_x \prec t_b$, and there does not exist such t_k, that $t_x \prec t_k \prec t_b$. 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 dependence 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 concurrently, and t_j should be executed after executing t_i, otherwise we will get wrong results.

D. Control Dependency

If a node exists in all execution paths, it is a *unavoidable node*. If a node t_j is not an unavoidable node, any node t_i whose outdegree is larger than 1 in the path from the start node to t_j is a *dominant node* of t_j, and t_j is *control dependent* on t_i, which is denoted by $t_i \rightarrow_c t_j$. In the example shown in Figure 1, $t_2 \rightarrow_c t_3$, $t_2 \rightarrow_c t_4$ and $t_2 \rightarrow_c t_5$. Control dependence relation is transitive. If $t_i \rightarrow_c t_j$ and $t_j \rightarrow_c t_k$ then $t_i \rightarrow_c t_k$.

We use $\rightarrow$ to denote data or control dependency when the concrete type of dependencies 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 $t_1 \rightarrow^* t_n$.

III. Theories of Recovery

A. 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 dependence relations and the control dependence relations among workflow tasks can further spread the damage to other data objects. We identify corrupted data objects, *incorrect data objects*, based on the following axiom.

Axiom 1: A task will generate *incorrect* data if, and only if, any of the following conditions is true.

1) The task codes are malicious or the task should not be executed.
2) The data objects that the task reads are incorrect.

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

Definition 2: Given normal tasks and recovery tasks, the recovery is *strict correct* if and only if the following conditions hold.

1) (*Completeness of recovery*) No incorrect data exists after the recovery.
2) (*Safety of recovery*) No incorrect data is generated while executing recovery tasks.
3) (*Safety of normal services*) No incorrect data is generated while executing normal tasks.
4) (*Consistency to the workflow specification*) The execution of normal tasks and recovery tasks does not violate the specification of 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 describes that execution of normal tasks should be clean. In other words, if a new task tries to read corrupted data from some unrecovered tasks, it should be suspended until the data is clean.

Note that Condition 1 does not imply Condition 2 and Condition 3. Condition 1 requires that the recovery should be complete while it implies nothing about the procedure of the recovery.

In this paper, we assume that if a task t_i is corrupted, we can remove its effects in the workflow system by executing a task $undo(t_i)$, which can be implemented by reading the last version of the data objects before the attack from the log of the workflow management system. To recover affected tasks, we need to re-execute them. We denote the re-execution of task t_i by $redo(t_i)$. t_i and $redo(t_i)$ are different executions of the same task. $redo(t_i)$ refers to the execution when carrying out the attack recovery.

B. Recovery Tasks

This section describes how to find undo and redo tasks. From Axiom 1 we can get the following theorem for undo tasks.

Theorem 1 (Undo tasks): Assume that $\mathcal{B}$ is the set of malicious tasks already known. Task t_j generates incorrect data and needs to be undone if, and only if, any of the following conditions are satisfied.

1) $t_j \in \mathcal{B}$
2) $\exists t_i \in \mathcal{B}, t_j \in \mathcal{L}, t_i \rightarrow_c^* t_j$, and $t_j \notin succ(redo(t_i))$
3) $\exists t_i \in \mathcal{B}, t_i \rightarrow_f^* t_j$
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 $t_k \in succ(redo(t_i))$

Proof Sketch: 1) *Sufficient condition*. In the first two conditions, t_j should not have been executed. It satisfies condition 1 of Axiom 1. In the third condition, t_j reads corrupted data. In the fourth condition, t_j reads some data objects that are not up to date. Thus, the last two conditions satisfy condition 2 of Axiom 1. Therefore, t_j generates incorrect data objects and should be undone. 2) *Necessary condition*. If a task t_j generates incorrect data objects, then its codes are malicious, it should not have been executed, or it reads incorrect data objects. In a workflow system, we can enumerate all possible conditions listed in the theorem. □

We explain the theorem by Figure 1. Task t_1 marked with 'B' was corrupted directly by attackers and is reported by the IDS, $\mathcal{B} = \{t_1\}$. The data that t_1 generates is corrupted, and t_1 needs to be undone, as indicated by condition 1 in Theorem 1.

In the figure, task $t_1 \rightarrow_f t_2$, where $t_1 \in \mathcal{B}$. t_2 is infected by task t_1 because it reads corrupted data from t_1 and then creates wrong results. Tasks t_4, t_8, and t_{10}, as described by condition 3 also create wrong results. Now, $\mathcal{B} = \{t_1, t_2, t_4, t_8, t_{10}\}$.

The situation described in condition 2 is shown by task t_3. The execution of task t_3 is based on the executing result of

task t_2, where $t_2 \in \mathcal{B}$. Since task t_2 is affected by t_1, it is possible that the choice of execution path is wrong. We must redo task t_2 and then check whether t_3 is still on the execution path: check if $t_3 \in succ(redo(t_2))$ in the recovery. If $t_3 \notin succ(redo(t_2))$ is in the recovery, then the data t_3 generated before is corrupted, and t_3 needs to be undone, although the computing of t_3 is correct.

For the last case described in Theorem 1, please refer to the execution of task t_6. 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 is executed along a new execution 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 recovery execution. Therefore, t_6 got a wrong result in the attacked execution, and the data that it generated was corrupted.

We call the tasks described by condition 2 and condition 4 as *candidate undo tasks* because we do not know if they really should be undone until $redo(t_i)$ is executed. If they need to be undone, then they are added to $\mathcal{B}$.

The tasks that have already been undone and are still on the re-executing path should be redone. We have the following theorem for redo tasks.

Theorem 2 (Redo tasks): Assume that $\mathcal{B}$ is the set of bad tasks already known and $t_i \in \mathcal{B}$, 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) $\exists t_j \in \mathcal{B}, t_j \rightarrow_c^* t_i, t_i \in succ(redo(t_j))$

PROOF SKETCH: In both cases, t_i has been damaged and is on the re-executing path. Thus, t_i needs to redo to meet the specification of workflows. □

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 be redone, as stated in case 1 of Theorem 2. Since neither task t_3 nor task t_4 is on the re-executing path of the workflow, they do not need to be redone according to Theorem 2. Redoing them does not meet the specification of the workflow because redoing them will generate corrupted data.

C. Partial Orders Caused by Dependence Relations

Since undo and redo tasks are not defined by the original Workflow, we must create partial orders among these tasks and normal workflow tasks to guarantee that our recovery is strict correct.

Theorem 3 (Partial orders among recovery tasks): Given any two tasks t_i and t_j and the system log $\mathcal{L}$, the recovery is strict correct only if the partial orders of the recovery tasks are derived by the following rules.

1) $t_i \prec t_j \Rightarrow redo(t_i) \prec redo(t_j)$
2) $t_i \rightarrow t_j \Rightarrow redo(t_i) \prec redo(t_j)$
3) $\forall t_i, undo(t_i) \prec redo(t_i)$
4) $t_i \rightarrow_a t_j \Rightarrow undo(t_j) \prec redo(t_i)$
5) $t_i \rightarrow_o t_j \Rightarrow undo(t_j) \prec undo(t_i)$
6) $t_i \rightarrow_c t_j, t_j \in succ(t_i) \Rightarrow redo(t_i) \rightarrow_c redo(t_j) \wedge redo(t_j) \in succ(redo(t_i))$
7) $t_i \rightarrow_c t_j, t_j \notin succ(t_i) \Rightarrow redo(t_i) \rightarrow_c redo(t_j) \wedge redo(t_j) \notin succ(redo(t_i))$
8) $t_i \in \mathcal{B}, t_j \in \mathcal{L}, t_i \rightarrow_c^* t_j$ and $t_j \notin succ(redo(t_i)) \Rightarrow redo(t_i) \rightarrow_c redo(t_j) \wedge undo(t_j) \in succ(redo(t_i))$
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 $t_k \in succ(redo(t_i)) \Rightarrow redo(t_i) \rightarrow_c undo(t_j) \wedge undo(t_j) \in succ(redo(t_i))$
10) $t_i \in \mathcal{B}, \exists t_j \in \mathcal{B}, t_j \rightarrow_c^* t_i, t_i \in succ(redo(t_j)) \Rightarrow redo(t_i) \rightarrow_c redo(t_j) \wedge redo(t_j) \in succ(redo(t_i))$

PROOF: See appendix.

In order to run both the recovery tasks and normal workflow tasks concurrently, we introduce partial orders among recovery tasks and normal tasks.

Theorem 4 (Partial orders about normal tasks): Given normal workflow tasks $\mathcal{N}$ and the system log $\mathcal{L}$, if every data object has only one copy, say, the value of a data object will be lost after writing, the recovery is strict correct only if then precedence relations are derived by the following rules.

1) $(t_i \rightarrow_f t_j) \vee (t_i \rightarrow_a t_j) \vee (t_i \rightarrow_o t_j) \vee (t_i \rightarrow_c t_j), t_j \in \mathcal{N} \Rightarrow undo(t_i) \prec redo(t_i) \prec t_j$
2) $t_i \rightarrow_c^* t_k, t_k \rightarrow_f^* t_j, t_k \notin \mathcal{L} \cup \mathcal{N}, t_j \in \mathcal{N} \Rightarrow undo(t_i) \prec redo(t_i) \prec t_j$

PROOF: See appendix.

The results of Theorem 4 is pretty reasonable. For example, we cannot expect a task that refers to the corrupted x to get correct results before x is repaired. If there exists such task, it should wait until x is recovered. Similarly, if a recovery task $redo(t_i)$ needs to read from y to repair x, then a normal task that writes to y is supposed to wait until $redo(t_i)$ is done. Otherwise, the recovery task $redo(t_i)$ will be corrupted.

Theorem 4 indicates that, to guarantee the strict correctness of recovery, a normal task cannot be executed before all recovery tasks are figured out. Unfortunately, we do not know the set of recovery tasks until the analysis of recovery tasks is complete. In other words, we cannot run any normal task until all malicious tasks reported by the IDS have been processed, which may cause temporary delay to process normal tasks when the attacking rate is high and the system is busy analyzing damages.

If no partial order is defined between two tasks, they can be run in any sequence without comprising the correctness of the execution results.

D. Recovery Strategies

When we build a recovery system, there are three possible strategies for recovery.

Strict correctness. Do the recovery while guaranteeing the correctness of executing both the recovery and normal tasks. We may delay normal tasks while damages are analyzed. In this paper, we adopt this strategy, in order to guarantee the correctness and termination of the recovery.

Obtain concurrency while taking risks of corrupting tasks. In this strategy, the system executes a task before knowing

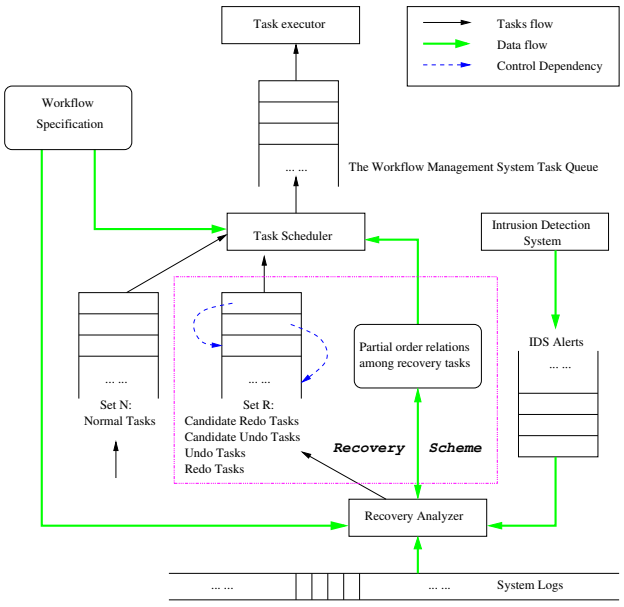


Fig. 2. Processing Structure of an Attack Recovery System

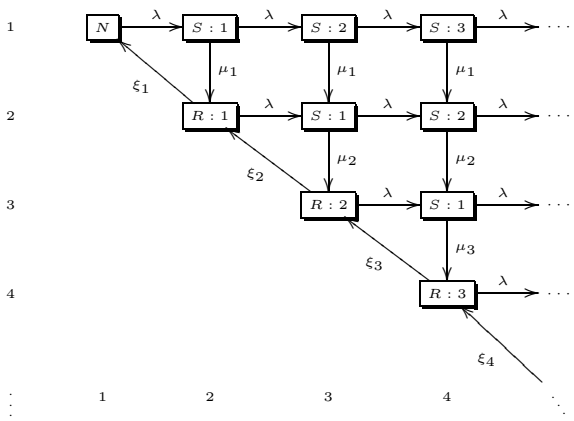


Fig. 3. State Transition Graph of the Recovery System

all dependence relations. However, as we mentioned before, both recovery tasks and normal tasks may be corrupted and we need to repair them again. This strategy in fact introduces more recovery tasks and costs, because the more tasks are executed, the more tasks may be corrupted. Even worse, we cannot guarantee the system will be repaired, since we cannot guarantee the recovery is correct and terminable.

Obtain concurrency while taking risks of corrupting only normal tasks. Theorem 4 is derived from the assumption that every data object has one copy. Multiple versions of data objects can break anti-flow and output dependence relations. If every data object has multiple versions, normal tasks can be executed without blocking while we guarantee the correctness of recovery. However, since the recovery is not complete, we cannot guarantee the correctness of executing normal tasks. Furthermore, multiple versions for each data object also introduce extra storage costs. We discuss this strategy in another paper.

IV. Recovery System and State Transition Model

A. Architecture of the Recovery Systems

The structure of the recovery system is shown in Figure 2. Our recovery system consists of an independent IDS to identify attacks, a recovery analyzer to evaluate damages of the system, and a scheduler to schedule both recovery tasks and normal tasks.

In the system, the IDS periodically reports intrusions to the system by putting 'IDS Alerts' in a queue. The recovery analyzer generates recovery tasks, works out related partial orders, and puts them in the queue of recovery tasks. The task scheduler schedules both recovery tasks and normal tasks according to their partial orders.

B. Implementation Issue

Our theories depend on data and control dependence relations that can be calculated when compiling workflows. Another important data structure is the system log, which exists in all workflow management systems. It is not difficult to design algorithms for the scheduler and the recovery analyzer. Thus, our system model is easy to implement and is practical.

C. State Transition of the System

The state transition graph (STG) of the system is shown in Figure 3. In the figure, we denote by 'N' the NORMAL state. 'S:n' represents the SCAN state with n IDS alerts in the queue, and 'R:n' represents the RECOVERY state with n units of recovery tasks in the queue, where 1 unit of recovery tasks corresponds to a set of tasks for repairing damages caused by 1 attack.

There are three categories of states of the attack recovery system: 'NORMAL', 'SCAN', and 'RECOVERY'.

In the NORMAL state, there is no intrusions reported in the system. The recovery analyzer does nothing and the scheduler executes normal tasks.

In the SCAN states, intrusions are reported, and the queue of IDS Alerts is not empty. The recovery analyzer analyzes damages to the system, and generates recovery tasks and partial orders among them. As we mentioned before, recovery may redo some tasks to repair damaged data objects. The redo tasks may read some data objects that new IDS alerts try to mark as damaged data objects, which in turn will corrupt recovery tasks. Therefore, in the SCAN state, recovery tasks may not be executed.

In the RECOVERY states, the queue of IDS alerts is empty. All damages of the system are identified. The scheduler schedules recovery tasks and new tasks according to their partial orders.

Although the structure in Figure 2 looks like a queuing network, according to the restrictions that the system does not execute recovery tasks in the SCAN state, which leads to that

the scan and the recovery cannot run in parallel, the system cannot be modeled by a queuing network.

The recovery system starts form the state NORMAL. It transits to the SCAN states whenever there are IDS alerts arrived. After all IDS alerts have been processed, the system goes to the RECOVERY states. The system returns to the NORMAL state when all recovery tasks have been executed.

If there are no further intrusions, the recovery will definitely be terminated because the system will transit to the RECOVERY states after all damages are analyzed, then transit to the NORMAL state after all recovery tasks are executed.

D. Parameters of the System

It is well known that intrusions occur sporadically, with long time periods where there are no successful attacks, interspersed with short bursts of multiple attacks. However, there is still no agreement about what probability distribution best describes the intrusions.

To obtain reasonable analytical results, we consider the continuous rate of intrusions to learn the response of our system when intrusions happen.

In our model, we assume that the arriving of IDS alerts is Poisson distribution. The probability of n IDS alerts arrived in $[0, t)$ is $P_n(t) = \frac{(\lambda t)^n}{n!} e^{-\lambda t}$. In other words, any S:$n$ state transits to S:$n+1$ with transition rate λ. The distribution function of IDS alerts is $F(t) = 1 - e^{-\lambda t}$. Thus, the probability density function of inter-arrival times of the IDS alerts is given by $f(t) = \lambda e^{-\lambda t}$, which is exponential distribution with parameter λ. Since an IDS alert causes one unit of recovery tasks, the transition direction always directs to the right side. We assume the processing time of IDS alerts and recovery tasks are exponential distribution with parameter μ_k and $\xi_k, 1 \leq k \leq \infty$, respectively, where $\mu_k = f(\mu_1, k)$ and $\xi_k = g(\xi_1, k)$. μ_k and ξ_k can also be considered as transition rates among states.

Since both the analyzer and scheduler need to check dependence relations to all items in queues, the more items in the queues, the more time will be spent. Say, $\mu_1 \geq \mu_2 \geq \cdots \geq \mu_k \cdots$, and $\xi_1 \geq \xi_2 \geq \cdots \geq \xi_k \cdots$, where $1 \leq k \leq \infty$. We use function f and g to simulate the degradation of performance when the number of items in queues increases. Given λ, μ_1, ξ_1, f and g, a model is solely determined.

Although the recovery system can find more damages than the IDS, the recovery still depends on the accuracy of the IDS. However, we assume that all corrupted tasks will ultimately be identified by the administrator of the system, even if they are not identified by the IDS. Since our system does not depend on timely reporting from the IDS, the delay of identifying a malicious task is not a problem. Therefore, we do not consider parameters of the IDS, such as false alarm rate and delay in this paper.

E. Fit Infinite States to a Real System

Our model has infinite states, which is not practical in the real world. A real system has limited resources so its buffer size for queues is limited. Therefore, the number of states of a real attack recovery system is limited. In fact, when a queue is full, no further state transition about the queue can be made. Hence, the number of total states is restricted.

Based on the assumptions about parameters and the above discussion, the state transition of our model becomes a finite states Continuous-Time Markov Chain (CTMC) [3], [4] that can be characterized by a *generator matrix* $\mathbb{Q} = (q_{i,j})$ and initial state probability vector $\underline{\pi}(0)$, where $q_{i,j}$ is the transition rate from i to j and $q_{i,i} = -\sum_{j \neq i} q_{i,j}$.

There are two important queues in the system: the queue of recovery tasks, and the queue of IDS alerts. The queue size of recovery tasks is critical to the performance of the system. Once the buffer of recovery tasks is full, no new IDS alerts can be processed because there is no space to store new recovery tasks. For example, let the buffer size of recovery tasks be 4. The STG for the system will look like the STG in Figure 3 except all parts beyond row 4 would be eliminated. When the buffer of recovery tasks is full (number of IDS alerts is 4), any new arrival IDS alerts cannot be processed. The recovery analyzer is simply blocked. As long as the recovery analyzer keeps on blocking, the queue of IDS alerts will fill up. After the queue of IDS alerts is full, it will lose IDS alerts. Therefore, all parts beyond column 4 are not helpful for improving the overall performance of the system. However, a larger buffer for IDS alerts can help to cache peak traffic. As long as the mean rate of the IDS alerts remains stable, the system can handle the situation of some peak traffics with a larger buffer of IDS alerts.

According to our discussion, the buffer size of recovery tasks determines the overall performance of a system. In this paper, an n size buffer of recovery tasks is modeled by a n rows by n columns STG.

As long as the queue of recovery tasks is full, the system will be at states at the right edge of STG, indicating that the system has limits, beyond which IDS alerts are lost by the recovery system. When and how long a system stays at the right edge of STG describes how many IDS alerts could be lost by the system. Given a probability distribution among all states of a STG, we define *loss probability* as follows.

Definition 3: Given an n states STG, a vector of probability distribution $\pi = (p_1, p_2, \cdots, p_n)$, and a set $\mathcal{E}$ of states at the right edge of STG, the loss probability of the STG with respect to π is $lp_\pi = \sum_{i \in \mathcal{E}} p_i$.

Loss probability describes whether a system reaches its limits under given condition π. If lp_π is very small, the system is prone to working very well. Otherwise, the system is prone to losing IDS alerts and failing to work efficiently.

Definition 4: If a system exists a steady state whose probability distribution is given by π, where $lp_\pi = \epsilon$, we say the system is ϵ-convergence.

ϵ-convergence describes how many IDS alerts could be lost at the steady state of a system. It is more practical for introducing features of a real system. A 1-convergence system is the worst system in theory and is useless in practice. The goal of designing a system is to let the ϵ as small as possible.

V. EVALUATION

In this section, we evaluate the recovery system while the attacks are on-the-fly. We focus on the state probability of the system, and the loss probability of the IDS alerts, because both of them reflect performance features of the system. For example, if the probability that the system stays at the NORMAL states is 0.1, the system will be busy in recovery and the system is not able to effectively process normal tasks. For example, if the probability that the system stays at the NORMAL states is 0.9, the system will spend little time on the recovery and spend more time on processing normal tasks. We evaluate not only the steady states, but also the transient states of the system. Therefore, the system responses to attacks in both sustained and burst fashion can be obtained from our analysis.

A. Steady-State Behaviors

The steady state of a system is the state that all features of the system do not change any more after running a long period of time. It may not exist at all for a specific CTMC. Fortunately, most real systems do have their steady states. Once a n by n generator matrix $\mathbb{Q}$ is given, the steady-state probability vector $\underline{\pi}$ is determined by Equation 1.

$$\underline{\pi}\mathbb{Q} = \underline{0}, \sum_{1 \le i \le n} \pi_i = 1. \quad (1)$$

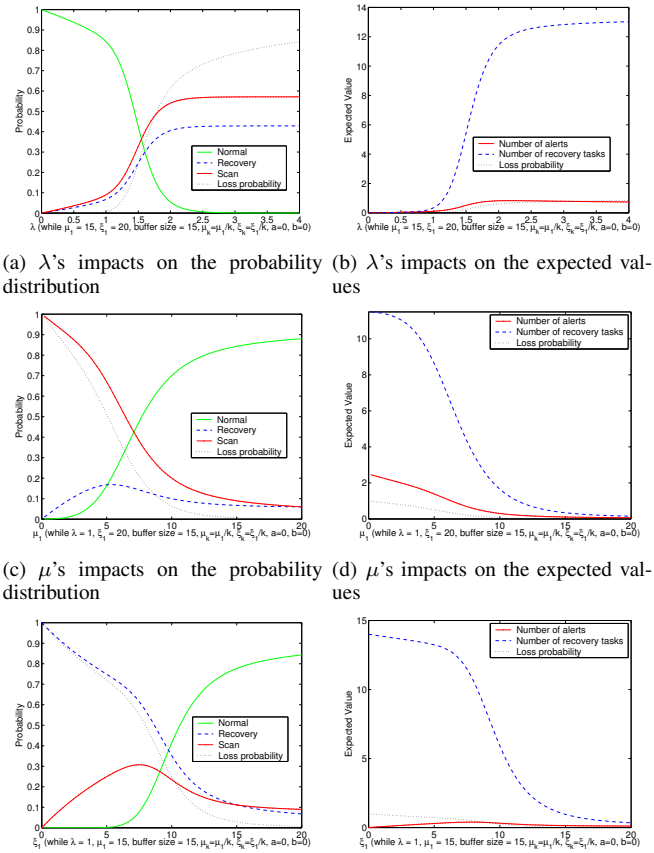
1) Impacts on the loss probability with different buffer size, f and g:

Case 1: $\lambda = 1, \mu_1 = 15$, and $\xi_1 = 20$, buffer size changes from 2 to 30. The results are shown in Figure 4 with different f and g.

Remark: If the speed of degradation of μ_k and ξ_k is very slow while k increases, loss probability can be reduced significantly by increasing the buffer size of IDS alerts, as shown in Figure 4(a). The condition does not hold in most real systems. The analyzer needs to check all dependence relations among existing recovery tasks to generate a correct recovery scheme after new IDS alert arrives. The checking time will be a function of the number of existing recovery tasks. It is more realistic that μ_k and ξ_k decreases while k increases. When the attenuation of μ_k and ξ_k is very quick and there are too much items in queues to process, the steady state of the system is prone to higher loss probability. Intuitively, when the buffer size increases not too much and the attenuation of μ_k and ξ_k is not too much, the loss probability decreases. If we allow the queues to be too large, the loss probability will increase due to significant degradation of processing speed, which can be found in Figure 4(b) and Figure 4(c). When μ_k decreases faster than ξ_k, the results are better than the contrary case, which can be found in Figure 4(d) comparing with Figure 4(c).

2) Impacts on steady-state probability with different λ, μ and ξ: For simplicity of comparison, we stick to the condition of $\mu_k = \mu_1/k, \xi_k = \xi_1/k$ and buffer size is 15 in this section.

Case 2: $\mu_1 = 15, \xi_1 = 20$, λ changes from 0 to 4. The probability distribution is shown in Figure 5(a). The expected values of number of items in queues are shown in Figure 5(b)[2].



(a) λ's impacts on the probability distribution (b) λ's impacts on the expected values

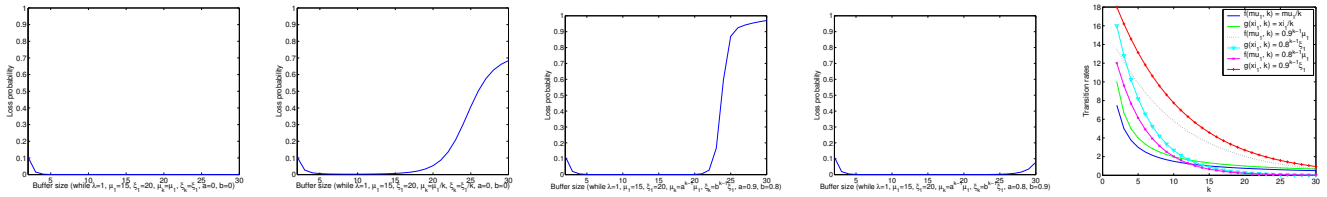
(c) μ's impacts on the probability distribution (d) μ's impacts on the expected values

(e) ξ's impacts on the probability distribution (f) ξ's impacts on the expected values

Fig. 5. Impacts on steady-state probability with different λ, μ and ξ

Remark: When the arrival rate of IDS alerts λ is less than 1, the system has high probability (> 0.8) to stay at the NORMAL state. The loss probability is very low. The expected values are also less than 1. The system can handle the situation very well. When λ is larger than 1, especially larger than 1.5, the loss probability and the probability of staying at the SCAN state increase very quickly, indicating that the system cannot handle all intrusions, and consequently, the performance of processing normal tasks degrades almost 100%. We can also observe that the queue of recovery tasks is full even though the expected number of IDS alerts is 1. The system cannot accept new IDS alerts to generate more recovery tasks although the buffer of IDS alerts is almost empty. Therefore, the buffer size of recovery tasks is a critical parameter for system performance. By checking the expected number of IDS alerts in the queue, we can reduce the buffer size for IDS alerts without compromising the loss probability of the system. In fact, we observed that the probability distribution on states that are near the right up conner of the STG usually are zeros if

[2]In Figure 5(b), Figure 5(d) and Figure 5(f) the loss probability is also drawn in the figures for reference. Please note that the value of loss probability is between 0 and 1, and it is not an expected value as read from label of y axis.



(a) $\mu_k = \mu_1$ and $\xi_k = \xi_1$ (b) $\mu_k = \frac{\mu_1}{k}$ and $\xi_k = \frac{\xi_1}{k}$ (c) $\mu_k = a^{k-1}\mu_1, \xi_k = b^{k-1}\xi_1, a = 0.9$ and $b = 0.8$ (d) $\mu_k = a^{k-1}\mu_1, \xi_k = b^{k-1}\xi_1, a = 0.8$ and $b = 0.9$ (e) Comparison of different f and g

Fig. 4. Impacts on the loss probability with different buffer size of IDS alerts, f and g

the system has low loss probability ($< 1\%$).

Case 3: $\lambda = 1, \xi_1 = 20$, μ_1 changes from 0 to 20. The probability distribution is shown in Figure 5(c). The expected number of items in queues are shown in Figure 5(d).

Case 4: $\lambda = 1, \mu_1 = 15$, ξ_1 changes from 0 to 20. The probability distribution is shown in Figure 5(e). The expected number of items in queues are shown in Figure 5(f).

Remark: Case 3 and Case 4 show that μ_1 and ξ_1 have similar effects on the system behaviors. When they are large enough, $e.g.\ > 15$, the system has high probability (> 0.8) to stay at the NORMAL state, which indicates that the degradation of performance for new arrival tasks is less than 20%. After exceeding a specific value, e.g. 15, μ_1 and ξ_1 has no significant impacts on improving the steady probability of the NORMAL. There exists a cost effective range of μ_1 and ξ_1, which can be observed in Case 3 and Case 4. Given a value of λ, the μ_1 and ξ_1 are supposed to be in the cost effective range for designing a new system.

B. Transient Behaviors

A transient state of a system is the state of the system at a specific time. The evaluation of transient states shows how quickly a system goes to its steady state, how much time is spent on each state, how many IDS alerts have been lost before the system enters its steady state, and so on. In fact, a system may satisfy us with its steady states, but disappoint us with its transient behaviors, e.g. taking too long to go to the steady states or loosing too many IDS alerts before it can effectively handle them. Transient behaviors also tells us what may happen if a system suffers a short term of high attacking rate.

Given a generator matrix $\mathbb{Q}$ and initial state probability vector $\underline{\pi}(0)$, transit state probability $\underline{\pi}(t)$ at time t is determined by Equation 2.

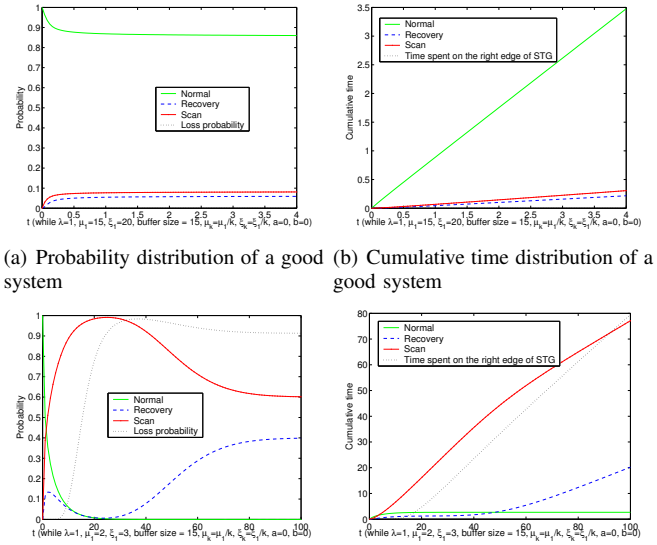
$$\frac{d}{dt}\underline{\pi}(t) = \underline{\pi}(t)\mathbb{Q} \tag{2}$$

Cumulative time $\underline{l}(t)$ spent on each state at time t is given by Equation 3.

$$\frac{d}{dt}\underline{l}(t) = \underline{l}(t)\mathbb{Q} + \underline{\pi}(0) \tag{3}$$

In this section, we stick to the condition of $\mu_k = \mu_1/k, \xi_k = \xi_1/k$ and buffer size $= 15$.

Case 5: $\lambda = 1, \mu_1 = 15, \xi_1 = 20$ and the system starts from the NORMAL state. We observe the behaviors of the system for 4 time-units. The probability distribution is shown in Figure 6(a). Cumulative time spent on each state is shown in Figure 6(b).



(a) Probability distribution of a good system (b) Cumulative time distribution of a good system

(c) Probability distribution of a poor system (d) Cumulative time distribution of a poor system

Fig. 6. Transient state probability

Remark: Case 5 shows what a good system looks like after it starts. Such systems enter their steady state very quickly and remain both low degradation of performance and low loss probability of IDS alerts in its steady state. Its loss probability of IDS alerts is almost not noticeable, given its proximity to the x axis and cannot be distinguished from the x axis in Figure 6(a). A system that has similar properties to Case 5 spends most of its time on executing normal tasks. Attacks are handled effectively and cost little time of the system. A real system is supposed to be designed in this style.

Case 6: $\lambda = 1, \mu_1 = 2, \xi_1 = 3$ and the system starts from the NORMAL state. We observe the behaviors of the system for 100 time-units. The probability distribution is shown in Figure 6(c). cumulative time spent on each state is shown in Figure 6(d).

Remark: Case 6 demonstrates what a system will be a given times during a period of rush hour attacks, or when the attacking rate is much higher than the value that a system is designed for. Under such conditions, the degradation of performance will be almost 100%, and there is no chance

to serve new arrival tasks. The loss probability goes up very quickly (< 30 time-units) and remains at the range from 0.9 to 1. Moreover, such systems spend most of their time on analysis and recovery. By observing time spent on the right edge of STG in Figure 6(d), the system spent about 80% time on the right edge of STG, indicating that the queue of recovery tasks is full.

An interesting fact is that $\mu_1 = 2$ and $\xi_1 = 3$ is already good enough for a small λ, such as $\lambda = 0.1$. If the system is designed for $\lambda = 0.1$ we cannot say the system shown in Figure 6(c) is a poor system simply based on results in Case 6. Since the attacking rate to the system is 9 times larger than the goal of its design. But we can learn transient behaviors of the system from Case 6 when attacking rate is much higher than what it is designed for. In the case, the system can resist such high attacking rate about 5 time-units if it is at the NORMAL state when the attacks start. After 5 time-units, the loss probability goes up. The results show how long the system can resist to a specific high attacking rate without compromising its loss probability.

VI. Guidelines for Designing a System

We summarize results of our evaluation and give guidelines for designing an attack recovery system with target parameter λ and ϵ step by step, where λ is the expected attacking rate that the system is designed to handle and the system is ϵ-convergence. In our analysis, we found that when the loss probability is low, the system always has a good state distribution probability. So, consider the loss probability is enough.

First, design and evaluate the performance degradation of analyzing algorithm and scheduling algorithm. Evaluate μ_k and ξ_k, where $1 \leq k \leq n$ and n is the maximum buffer size of recovery tasks that we want to try, e.g. $n = 30$. The performance degradation while k increases should be designed as slow as possible. Otherwise, the loss probability is very sensitive to the change of buffer size of recovery tasks.

Second, increase the buffer size of recovery tasks from 2 to n, or until the loss probability begins to increase. Check if ϵ can be satisfied in the range of low loss probability. If so, choose a buffer size for recovery tasks. Otherwise, redesign all algorithms and repeat the procedure.

There are totally two ways to reduce the loss probability of a system. One way is to improve μ_1 and ξ_1, say, the speed of processing recovery tasks and IDS alerts when there are only one IDS alert and one unit of recovery tasks in buffers. The other way is to improve algorithms with less complexity to slow down the decreasing speed of μ_k or ξ_k or both while k increases, and increase corresponding buffer size. Select the one which costs less.

Finally, the buffer size of IDS alerts may be less than the buffer size of recovery tasks according to its expected value. But we do not recommend to do so since the expected value is for considering a long period of time. The buffer of IDS alerts may be overflowed by a transient high attacking rate. To reduce the buffer size of IDS alerts is worthless since it saves little space.

Bigger buffer size of IDS alerts may help the system to handle transient high attacking rates, but, it is not a long term solution. After the queue of recovery tasks is full, the queue of IDS alerts will fill up very quickly if the system suffers a long term of high attacking rate. Therefore, design the buffer size of IDS alerts according to the peak rate the system wants to handle. This could be achieved by inspecting transient behaviors of the system while applying desired peak attacking rate to its steady state.

VII. Related Work

The work most similar to ours handles malicious transactions in the database system, as discussed in [5]. 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.it is unable to trace damage spreading and cannot locate all damages to the system. In contrast, we show that to guarantee the correct recoveries of workflow ,we need all data and control dependence relations among transactions. Otherwise, both recoveries and newly executed transactions could be corrupted.

The failure handling of workflow has been discussed in recent work [6], [7], [8]. Failure handling is different with attack recovery in two aspects. On one hand, they have different goals. Failure handling tries to guarantee the atomic of workflows. When failure happens, their work find out 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 failure malicious tasks have no effects 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 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 can do nothing because no failure occurred. Attack recovery is supposed to remove the effects of malicious tasks after they are committed.

Rollback recovery, e.g. [9], [10], is surveyed in [11]. 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 dependence relations inside workflow tasks. In fact, message passing is a kind of data dependence relation but not vice versa (e.g., a data dependence relation caused by more than one message passing steps or by sharing data). We also observed that in workflow recovery an execution path may change due to control dependence, causing different patterns of message passing. In addition, our methods exploit more detail in dependence relations than the methods that are message passing based; therefore our method is more effective and efficient

for workflow recovery. Our method also better matches the workflow model.

De-centralized workflow processing is becoming more and more popular. In distributed workflow models, workflow specifications cannot be accessed in a center node. They are carried by workflow itself or stored in a distributed style. In either case, our theories are still practical. We need to process the specifications of workflow in a distributed style.

In some work such as [12], security and privacy is Important, and the whole specification of workflows avoids being exposed to all processing nodes to protect privacy. Our theories are based on the dependence relations among tasks. The specification can be best protected by exposing only dependent relations to the recovery system.

VIII. Conclusions

We described fundamental theories for on-line attack recovery of workflows. While an independent IDS reports malicious tasks periodically, our techniques find all damages caused by the malicious tasks and repair them automatically. We described the architecture of the recovery system and analyzed its performance when intrusions happen. Both steady states and transient states are analyzed. The analytical results show that our system is practical.

References

[1] 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)*, vol. 3, no. 4, pp. 227–261, 2000.

[2] J.-L. Lin and M. H. Dunham, "A survey of distributed database checkpointing," *Distributed and Parallel Databases*, vol. 5, no. 3, pp. 289–319, 1997.

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

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

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

[6] J. Eder and W. Liebhart, "Workflow recovery," in *Conference on Cooperative Information Systems*, 1996, pp. 124–134.

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

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

[9] D. R. Jefferson, "Virtual time," *ACM Transaction on Programming Languages and Systems*, vol. 7, no. 3, pp. 404–425, July 1985.

[10] Y. bing Lin and E. D. Lazowska, "A study of time warp rollback machanisms," *ACM Transactions on Modeling and Computer Simulations*, vol. 1, no. 1, pp. 51–72, January 1991.

[11] 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*, vol. 34, no. 3, pp. 375–408, September 2002.

[12] 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*. ACM Press, 2001, pp. 48–57.

Appendix

Proof of Theorem 3

1) Comes directly from criterion 4 of Definition 2.
2) Derived from rule 1 and the definition of relation $\rightarrow$
3) By contradiction. If $redo(T_i) \prec undo(T_i)$ then the effects of task T_i will be undone, which violates criterion 4 of Definition 2.
4) By contradiction. Assume $redo(T_i) \prec undo(T_j)$. Since $T_i \rightarrow_a T_j, R(redo(T_i)) = R(T_i)$ and $W(undo(T_j)) = W(T_j)$, so $R(redo(T_i)) \cap W(undo(T_j)) \neq \phi$. Moreover, $W(undo(T_j))$ is corrupted before $undo(T_j)$. Therefore $redo(T_i)$ reads corrupted data from $R(redo(T_i)) \cap W(undo(T_j))$ then generates corrupted data, which violates criterion 2 of Definition 2.
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 older version than $W(T_j)$ for $W(T_i) \cap W(T_j)$. If $undo(T_i) \prec undo(T_j)$ 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 2.
6) Comes directly from criterion 4 of Definition 2.
7) Comes directly from criterion 4 of Definition 2.
8) Comes directly from condition 2 of Theorem 1.
9) Comes directly from condition 4 of Theorem 1.
10) Comes directly from condition 2 of Theorem 2. □

Proof of Theorem 4

1) Comes directly from criterion 4 of Definition 2.
2) $undo(T_i) \prec redo(T_i)$ come directly from rule 3 of Theorem 3. If T_j is data dependent on T_i we prove the result by contradiction. Since $R(redo(T_i)) = R(T_i)$ and $W(redo(T_i)) = W(T_i)$ 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 $W(redo(T_i)) \cap R(T_j)$ the $redo(T_i)$ has not created it. So the task T_j gets wrong data and be corrupted.
   - $redo(T_i) \rightarrow_o T_j$. After executing T_j, $redo(T_i)$ writes $W(redo(T_i)) \cap W(T_j)$ again. So the executing results of $redo(T_i)$ in $W(redo(T_i)) \cap W(T_j)$ 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 $R(redo(T_i)) \cap W(T_j)$. But according to the definition of workflow, $redo(T_i)$ should read data that exists in $R(redo(T_i)) \cap W(T_j)$ 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 2, or the recovery task $redo(T_i)$ is corrupted, which violates criterion 2 of Definition 2.
   If T_j is control dependent on T_j then the execution of T_j depends on the executing result of T_i. If $T_j \prec redo(T_i)$ then it is possible that $T_j \notin succ(redo(T_i))$ after $redo(T_i)$ is done. In this case, T_j creates corrupted data according to the Theorem 1 therefore the execution of T_j violates both criterion 3 and criterion 4 of Definition 2.
3) $undo(T_i) \prec redo(T_i)$ come directly from the rule 3 of Theorem 3.
   We prove $redo(T_i) \prec T_j$ by contradiction. Since $redo(T_i) \in \mathcal{R}$ is not done we do not know if $T_k \in succ(redo(T_j))$. Assume $T_j \prec redo(T_i)$. After the $redo(T_i)$ is done, it is possible that $T_k \in succ(redo(T_j))$. According to condition 4 of Theorem 1, T_j should be undone because it creates corrupted data, which violate criterion 3 of Definition 2. □