

Recovering from Malicious Attacks in Workflow Systems

Yajie Zhu, Tai Xin, and Indrakshi Ray

Department of Computer Science
Colorado State University
{zhuy,xin,iray}@cs.colostate.edu

Abstract. Workflow management systems (WFMS) coordinate execution of logically related multiple tasks in an organization. Such coordination is achieved through dependencies that are specified between the tasks of a workflow. Often times preventive measures are not enough and a workflow may be subjected to malicious attacks. Traditional workflow recovery mechanisms do not address how to recover from malicious attacks. Database survivability techniques do not work for workflow because tasks in a workflow have dependencies that are not present in traditional transaction processing systems. In this paper, we present an algorithm that shows how we can assess and repair the effects of damage caused by malicious tasks. Our algorithm focuses not only on restoring the consistency of data items by removing the effects of malicious tasks but also takes appropriate actions to ensure the satisfaction of task dependencies among all the committed tasks.

1 Introduction

Workflow management systems (WFMS) are responsible for coordinating the execution of multiple logically related tasks performed by an organization. Since vulnerabilities cannot be completely removed from a system and preventive measures sometimes fail, a workflow may be subject to malicious attacks. A malicious attacker may create an illegal task or corrupt a task in a workflow to gain some personal benefits. This malicious task would possibly corrupt data items accessed by some benevolent tasks, or it may trigger some other tasks in this workflow due to the existence of intra-task dependencies in the workflow. Further, tasks that are dependent upon this malicious task can, in turn, corrupt other data items and affect other tasks. The process may continue and the damage can spread in a short span of time. In this paper we present an algorithm that shows how a workflow can detect and recover from such malicious attacks.

Recovering from malicious attacks have been investigated in the context of database systems. In such systems, a transaction executed by a malicious user might corrupt some data item. Other transactions reading from this data item and writing on other data items help spread the damage. Ammann et al. [1] have proposed techniques for assessing and repairing such damage. Their techniques involve parsing the database log to check which transactions were affected by malicious transaction and undoing and redoing the affected transactions. Panda et al. [7] have also proposed a number of algorithms on damage assessment and repair; some of these store the dependency information in

separate structures so that the log does not have to be traversed for damage assessment and repair.

Techniques for damage assessment and recovery in database systems are not adequate for workflows. This is because transactions in a database are independent entities. The only way in which one transaction depends on another is through read-write dependencies. A workflow consists of tasks that have control-flow and data-flow dependencies specified among them, in addition to read-write dependencies. These dependencies ensure the proper co-ordination and execution of tasks in a workflow. The presence of these dependencies requires new techniques for damage assessment and recovery. In this paper, we propose one such technique.

A naive solution undoes the workflow with malicious tasks and the other workflows that have executed after it, and then re-executes these other workflows again. Our solution tries to improve upon this by minimizing the number of other workflows that need to be undone and re-executed. We take into account the nature of the dependencies that exist between the tasks of a workflow when we are doing damage assessment. The dependencies enable us to find out which specific tasks of the other workflows are affected. We undo these affected tasks only and re-execute them. Minimizing the number of tasks that are undone and re-executed speeds up the damage recovery process.

The remainder of the paper is organized as follows. Section 2 discusses related works in this area. Section 3 presents our definition of workflow and the various types of dependencies that exists in a workflow. Section 4 enumerates the information required and the assumptions in our recovery algorithm. Section 5 presents the workflow recovery algorithm. Section 6 concludes the paper with pointers to future works.

2 Related Works

Although a lot of research appears in workflow, we focus our attention to those discussing workflow dependencies, workflow recovery and workflow survivability. Eder and Liebhart [4] classify workflow as document-oriented workflow and process-oriented workflow, identify potential different types of failure, and discuss some possible recovery mechanisms. All these concepts are used to achieve one goal, which is to restore the most recent consistent process state after a failure, so that as little as possible long-duration work is lost and process execution can continue.

The FlowBack model [6] discusses the use of compensation for partial backward recovery of workflows. When user wants to abort the original process, the compensation process will be executed. The FlowBack prototype focuses on the flow control between tasks in the case of semantic failure but this flow control is ensured manually. The authors state that the compensation process can be very complex, because it must consider all the paths leading from each task in a workflow. Many tasks cannot be compensated due to the semantics of applications. Moreover, the compensation process is not transparent to the user.

Survivability has received attention in the database context. Ammann et al. propose a two pass repair algorithm for traditional database systems in their paper [1]. The static algorithm composes of two passes. Pass one scans the log forward from the entry where the first malicious task starts to locate every malicious and suspect tasks. Pass two

goes backward from the end of the log to undo all malicious and suspect tasks. They also proposed a dynamic repair algorithm. Their paper focuses on the purely syntactic information about the interleaving of read and write operations. Their algorithms cannot be applied to workflow systems having control flow and data flow dependencies.

Gore and Ghosh [5] discuss the recovery and rollback problem in distributed extended transactions. They propose a solution to the recovery problem using partial rollbacks. In the proposed model, the transactions communicate and collaborate only by exchanging messages. These messages are stored in message logs and message tables which are used extensively during recovery. The drawback of this approach is that it is not general – it is based on specialized log structures. Moreover, the authors do not address the issue of transaction dependencies in this paper.

Yu, Liu and Zang [8] describe an algorithm for on-line attack recovery of workflows. The algorithm tries to build the list of redo and undo tasks, after an independent Intrusion Detection System reports malicious tasks. They also relax the restriction of executing order that exist in an attack recovery system; they introduced multi-version data objects to reduce unnecessary blocks in order to reduce degradation of performance in recovery. The authors in this paper treat all types of control-flow dependencies in the same manner. In our paper, we show that the different types of control-flow dependencies require different treatment for recovery. Our algorithm takes into account the type of dependencies in performing recovery from malicious attacks.

3 Our Workflow Model

In our model, a “workflow” is a set of tasks with dependencies specified among them that achieve some business objective. Formally, a *workflow* $W_i = \langle T, D, C \rangle$ where T is the set of tasks in the workflow W_i , D is the set of dependencies, and C is the set of completion sets in T . A workflow W_i is said to be completed if all the tasks in anyone completion set are committed and all other tasks are either in an unscheduled or aborted state. We assume that each task in a workflow is a transaction as per the standard transaction processing model [2]. A task T_{ij} , which belongs to a specific workflow W_i , consists of a set of data operations (read or write) and task primitives; the begin, commit and abort primitives are denoted by b_{ij} , c_{ij} , and a_{ij} respectively. A task T_{ij} can be in any of the following *states*: *unscheduled* (un_{ij}), *initiation* (in_{ij}), *execution* (ex_{ij}), *prepare* (pr_{ij})(means prepare to commit), *committed* (cm_{ij}) and *aborted* (ab_{ij}). Execution of task primitives causes a task to change its state. Detailed state transition diagrams are shown in 1.

In order to properly coordinate the different tasks in a workflow system, dependencies are specified on task primitives, task operations, and task input/outputs. We refer to these different kinds of dependencies as *task dependencies*. The only kind of inter-workflow dependency we consider is *read-write dependency*. Between tasks of the same workflow we can have read-write dependency as well as *control-flow dependencies*, and *data-flow dependencies*.

A *control-flow dependency* specified between a pair of tasks T_{ij} and T_{ik} expresses how the execution of primitives (begin, abort and commit) of task T_{ij} relates to the execution of the primitives (begin, abort and commit) of another task T_{ik} . we give some

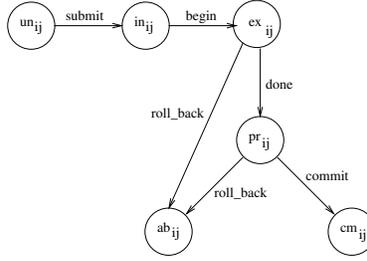


Fig. 1. States of Task T_{ij}

typical control-flow dependencies found in workflows. For a complete list of task dependencies, we refer the interested readers to the work by Chrysanthis 1991 [3].

- [Commit dependency]** $(T_{ij} \rightarrow_c T_{ik})$: If both T_{ij} and T_{ik} commit then the commitment of T_{ij} precedes the commitment of T_{ik} .
- [Strong commit dependency]** $(T_{ij} \rightarrow_{sc} T_{ik})$: If T_{ij} commits then T_{ik} must commits.
- [Abort dependency]** $(T_{ij} \rightarrow_a T_{ik})$: T_{ik} must abort if T_{ij} aborts.
- [Termination dependency]** $(T_{ij} \rightarrow_t T_{ik})$: T_{ik} cannot commit/abort until T_{ij} commits/aborts.
- [Force-commit-on-abort dependency]** $(T_{ij} \rightarrow_{fca} T_{ik})$: T_{ik} must commit if T_{ij} aborts.
- [Exclusion Dependency]** $(T_{ij} \rightarrow_{ex} T_{ik})$: if T_{ij} commits and T_{ik} has begun executing, then T_{ik} must abort.
- [Begin dependency]** $(T_{ij} \rightarrow_b T_{ik})$: T_{ik} cannot begin until T_{ij} has begun.
- [Begin-on-commit/abort dependency]** $(T_{ij} \rightarrow_{bc/ba} T_{ik})$: T_{ik} cannot begin until T_{ij} commits/aborts.
- [Force begin-on-begin/commit/abort/terminate dependency]** $(T_{ij} \rightarrow_{fbb/fbc/fba/fbt} T_{ik})$: T_{ik} must begin if T_{ij} begins/commits/aborts/terminates.

Task T_{ij} is *data-flow dependent upon* task T_{ik} in a workflow, denoted by $T_{ik} \rightarrow_{df} T_{ij}$, if there exists a data item x which is an input data item for task T_{ij} and an output data item of task T_{ik} , task T_{ij} accept x as an input from task T_{ik} , and both tasks belong to the same workflow. Note that if there is a data-flow dependency between task T_{ik} and T_{ij} , there will also be a control flow dependency of the form *begin-on-commit* between the two tasks. This is because for the output of T_{ik} to be available to T_{ij} , T_{ik} must commit before T_{ij} starts execution. So we do not consider data-flow dependencies separately while performing recovery.

A task T_{ij} is *read-write dependent upon* task T_{kl} if there exists a data item x such that: (i) T_{ij} reads x after T_{kl} has updated x , T_{kl} does not abort after T_{ij} reads x , and, (ii) if any T_{pq} updates x after T_{kl} has updated x but before T_{ij} reads it, then T_{pq} is aborted.

Example 1. Workflow W_1 consists of a set of tasks $T = \{T_{10}, T_{11}, T_{12}, T_{13}, T_{14}, T_{15}\}$. Each task performs a work: T_{10} – Make a car reservation from Company B; T_{11} – Reserve a ticket on Airlines A; T_{12} – Purchasing the Airlines A ticket; T_{13} – Canceling the reservation; T_{14} – Reserving a room in Resort C; T_{15} – Cancel the car reservation. The set of task dependencies D include: control-flow dependencies $\{T_{14} \rightarrow_a T_{10},$

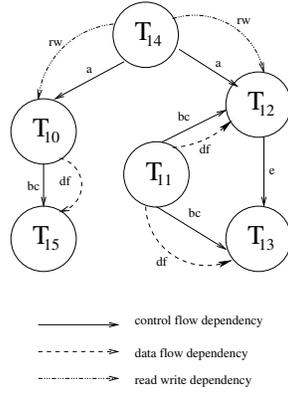


Fig. 2. Tasks and Dependencies in Workflow given in Example 1

$T_{11} \rightarrow_{bc} T_{12}$, $T_{11} \rightarrow_{bc} T_{13}$, $T_{12} \rightarrow_{ex} T_{13}$, $T_{14} \rightarrow_a T_{12}$, $T_{10} \rightarrow_{bc} T_{15}$, data Flow dependencies $\{ T_{10} \rightarrow_{df} T_{15}, T_{11} \rightarrow_{df} T_{12}, T_{11} \rightarrow_{df} T_{13} \}$, read-write Dependencies $\{ T_{14} \rightarrow_{rw} T_{10}, T_{14} \rightarrow_{rw} T_{12} \}$, As shown in this example, the three kinds of dependencies can co-exist in a workflow. The workflow can have several completion sets. Some possible completion sets are $\{ T_{14}, T_{10} \}$, $\{ T_{14}, T_{11}, T_{12} \}$, $\{ T_{14}, T_{13} \}$, $\{ T_{10}, T_{15} \}$ and $\{ T_{14}, T_{11}, T_{13} \}$.

4 Information Needed by the Repair Algorithm

Workflow repair is much more complex than the recovery in traditional transaction processing system. The repair process will need to know the state of the workflow after some malicious attacks (damage assessment) and it will also need to know the actions needed to perform the recovery. The specific actions to be taken depends on the structure of the workflow, that is, the dependencies associated with the workflow. The structure of the workflow can be obtained from the workflow schema. A *workflow schema* defines the type of a workflow. The specification of a workflow schema includes: the specification of tasks, the dependencies between these tasks, and the set of completion sets for this type of workflows. Each workflow is actually an instance of a workflow schema. We denote the schema associated with workflow W_i as WS_i . The information about all the workflow schema is maintained in stable storage.

In order to recover from a workflow system failure and/or malicious attack, the state information of a workflow need to be logged onto some stable storage. We propose that such information be stored in the system log. Execution of a workflow primitive, a task primitive, or a task operation results in the insertion of log a record. Execution of a begin primitive in a workflow results in the insertion of the following log record. $\langle START W_i, WS_i \rangle$ where W_i indicates the workflow id and WS_i indicates the schema id that corresponds to the workflow W_i . The completion of the workflow is indicated by a log record $\langle COMPLETE W_i \rangle$. Execution of the primitive begin for task T_{ij} results in the following records being inserted in the log: $\langle START T_{ij} \rangle$ where T_{ij}

is the task id. Similarly, $\langle COMMIT T_{ij} \rangle$ or $\langle ABORT T_{ij} \rangle$ records the execution of the primitive commit or abort. Execution of operations also cause log records to be inserted: A read operation log record $\langle T_{ij} X, value \rangle$ and a write operation log record $\langle T_{ij} X, v, w \rangle$.

5 Workflow Repair Algorithm

We focus on those intrusions that inject malicious tasks into the workflow management systems. Under these intrusions, the attackers can forge or alter some tasks to corrupt the system or gain some personal benefit. We assume that there is an Intrusion Detection Manager (IDM) in the system. The IDM can identify the malicious attacks and report the malicious tasks periodically. But it cannot discover all the damage done to the system. The damage directly caused by the attacker can spread into the whole workflow system by executing normal tasks without being detected by the IDM.

We explain some of the terms that we use to explain the algorithm. A *malicious task* is a committed task which is submitted by an attacker. A *malicious workflow* is one which contains at least one malicious task in any completion set of it. A task T_{ij} is *control-flow dependent upon* another task T_{ik} if the execution of task T_{ij} depends upon the successful/unsuccessful execution of task T_{ik} . Note that not all tasks related by control-flow dependencies are control-flow dependent on each other. Some control-flow dependencies impose an execution order on the different tasks; these do not require one task to be executed because of the successful/unsuccessful execution of another. Such tasks are not control-flow dependent on the other. For example, the commit dependency $T_{wi} \rightarrow_c T_{wj}$ controls the order of two tasks entering the commit state. It does not require any task T_{wi} (T_{wj}) to be executed because of another T_{wj} (T_{wi}).

For any control-flow dependency $T_{wi} \rightarrow_{d_x} T_{wj}$ between T_{wi} and T_{wj} , either task T_{wi} or task T_{wj} may be malicious/affected after a malicious attack taken place. For each case, we analyze how the malicious/affected task affects the other task. The cases when T_{wj} is control-flow dependent on a malicious or affected task T_{wi} are enumerated in Table 1. This table also gives the repair actions that are needed for task T_{wj} . Only two control-flow dependencies can cause task T_{wj} to be control-flow dependent upon T_{wi} : abort dependency and begin-on-commit dependency. The abort dependency requires T_{wj} to abort if T_{wi} aborts. Since T_{wi} is a malicious/affected task it must be undone. This necessitates undoing T_{wj} if it has already been committed. The begin-on-commit dependency ensures that T_{wj} will not begin until T_{wi} commits. Thus undoing T_{wi} requires an undo of T_{wj} if T_{wj} has been committed.

If T_{wj} is a malicious or affected task, it must be undone. Table 2 shows whether T_{wi} is control-flow dependent upon task T_{wj} and the repair actions needed for T_{wi} . In this case only T_{wi} is control-flow dependent upon T_{wj} if there is a strong-commit dependency. This dependency will be violated if we undo T_{wj} . In such a case, if T_{wi} has committed, it must be undone.

Before presenting our algorithm, we state our assumptions. (i) We consider the effect of committed malicious tasks in repairing. (ii) We assume that the execution of different workflows can interleave with each other. (iii) Each task is executed as a transaction which means it has the properties of atomicity, consistency, isolation, and durability

Dependency	T_{wj} CF dependent upon T_{wi} ?	Action in repair
$T_{wi} \rightarrow_{c/sc/t} T_{wj}$	No	No
$T_{wi} \rightarrow_a T_{wj}$	Yes	if T_{wj} committed, undo T_{wj}
$T_{wi} \rightarrow_{bc} T_{wj}$	Yes	if T_{wj} committed, undo T_{wj}
$T_{wi} \rightarrow_{b/ba} T_{wj}$	No	No
$T_{wi} \rightarrow_{fbb/fbc/fba/fbt} T_{wj}$	No	No
$T_{wi} \rightarrow_{fca/ex} T_{wj}$	No	No

Table 1. Is T_{wj} control-flow dependent upon the malicious or affected task T_{wi} ?

Dependency	T_{wi} CF dependent upon T_{wj} ?	Action in repair
$T_{wi} \rightarrow_{c/t/a} T_{wj}$	No	No
$T_{wi} \rightarrow_{sc} T_{wj}$	Yes	if T_{wi} committed, undo T_{wi}
$T_{wi} \rightarrow_{b/bc/ba} T_{wj}$	No	No
$T_{wi} \rightarrow_{fbb/fbc/fba/fbt} T_{wj}$	No	No
$T_{wi} \rightarrow_{fca/ex} T_{wj}$	No	No

Table 2. Is T_{wi} control-flow dependant upon the malicious or affected task T_{wj} ?

[2]. (iv) To ensure strict and serializable execution, we use the strict two phase-locking mechanism [2].

We denote the committed malicious tasks in a workflow system history by the set B , which are detected by IDM. Based on these malicious tasks, we can identify the corresponding malicious workflows as the set BW . The basic idea is that all malicious workflows must be undone. We must identify all affected task of other good workflows, and remove the effects of all malicious workflows.

Our algorithm proceeds in four phases. The first phase undoes all malicious workflows. It also collects the set of committed tasks for the good workflows. The second phase performs the damage assessment caused by malicious workflows by identifying all the set of affected tasks. In this we first identify the completed workflows that do not need any recovery action. We then identify all the tasks that were affected due to the presence of control-flow and read-write dependencies. The third phase undoes all the affected tasks. The fourth phase is responsible for re-execution and continuation of incomplete workflows.

Algorithm 1

Workflow Repair Algorithm

Input: (i) the log, (ii) workflow schemas, (iii) BW – set of malicious workflows

Output: a consistent workflow state in which the effects of all malicious and affected tasks are removed

Procedure WorkflowRepair

Phase 1: Undo malicious workflows and Identify the committed tasks sets of other workflows

$globalAborted = \{\}$ /* set holding aborted tasks of all workflows */

$committed[w] = \{\}$ /* set holding committed tasks of workflow W_w , which is not a malicious workflow */

```

workflowList = {} /* set holding the committed tasks of all but malicious workflows */
begin
  /* Scan backwards until we reach < START Wi > where Wi is the earliest malicious workflow*/
  do
    switch the last unscanned log record
    case the log record is < ABORT Twi >
      globalAborted = globalAborted ∪ {Twi}
    case the log record is < COMMIT Twi >
      if Ww ∉ BW ∧ Twi ∉ globalAborted
        for Ww, committed[w] = committed[w] ∪ {Twi}
    case the log record is update record < Twi,x,v,w >
      if Ww ∈ BW ∧ Twi ∉ globalAborted
        change the value of x to v /*undo the task in the malicious workflow*/
    case the log record is < START Twi >
      if Ww ∈ BW ∧ Twi ∉ globalAborted
        write < ABORT Twi > log record
    case the log record is < START Ww >
      if Ww ∈ BW
        write < ABORT Ww > log record
      else
        workflowList = workflowList ∪ committed[w]
  end //phase 1
  Phase 2: find all affected tasks
  corrupted = {} //a set holds all corrupted data items
  undo = {} //a set holds all affected tasks, which need to be undone
  finished = {} //a set holds all completed workflows
  begin
    /* Scan forward from < START Wi > where Wi is the earliest malicious workflow */
    do
      while not end of log
        switch next log record
        case log record = < COMPLETE Wi >
          if there exists a completion set Cj of Workflow Wi such that Cj ⊆ committed[i]
            finished = finished ∪ {i}
          else
            if i ∈ finished
              finished = finished - {i}
        case log record = < Tij,X,v,w > //write record
          /*find out the corrupted data item that is written by malicious or affected task*/
          if (Wi ∈ BW ∧ Tij ∉ globalAborted) ∨ Tij ∈ undo
            corrupted = corrupted ∪ {X}
        case log record = < Tij,X,value > //read record
          /*find out the affected task which reads a corrupted data item*/
          if X ∈ corrupted ∧ Wi ∉ BW ∧ Tij ∉ globalAborted ∧ Tij ∉ undo
            undo = undo ∪ {Tij}
      end while
    end do
  end begin
end begin

```

```

        /* get the set of control-flow affected tasks */
        newSet = getControlflowAffected( $T_{ij}$ , committed[i])
        if newSet  $\neq$  NULL
            undo = undo  $\cup$  newSet
            scan back to the earliest affected task in newSet
        case log record = < COMMIT  $T_{ij}$  >
            if  $T_{ij} \in$  undo
                committed[i] = committed[i] -  $T_{ij}$ 
    end //phase 2
Phase 3: undo all the tasks in the undo set.
begin scan backwards from the end of the log and undo all tasks in undo list
    do
        switch the last unscanned log record
        case log record = <  $T_{ij}, X, u, v$  >
            if  $T_{ij} \in$  undo
                restore the value of  $X$  to  $u$  //restoring before image
        case log record = < START  $T_{ij}$  >
            if  $T_{ij} \in$  undo
                write < ABORT  $T_{ij}$  > log record
    end //phase 3
Phase 4: resubmit the incomplete workflow to scheduler and continue the e execution.
begin
    for each committed[i]  $\in$  workflowList
        if  $i \notin$  finished
            submit committed[i] to the scheduler
    end //phase 4

```

6 Conclusions and Future Work

In this paper we have focussed on how to repair attacks caused by one or more malicious tasks in a workflow. In addition to the read-write dependencies that are present in the traditional transactions, workflows have control-flow and data-flow dependencies as well. These dependencies help spread the damage caused by malicious tasks and complicates the recovery process. Our algorithm removes the effects of workflows having malicious tasks and tries to minimize undoing the good tasks. Only good tasks that were affected are undone and re-executed.

We have given an algorithm that shows how the workflow can be repaired in the event of a malicious attack. A lot of work remains to be done. For instance, we need to formalize what we mean by a correct execution and correct repair of a workflow. Finally, we need to prove that our algorithm satisfies the correctness criteria. Workflow is an example of an extended transaction model. This work can be applied to other extended transaction processing models since read-write and control-flow dependencies also exist in these models. Specifically, we plan to propose how recovery from malicious transactions can occur in other kinds of extended transaction model.

Acknowledgement

This work was partially supported by NSF under Award No. IIS 0242258.

References

1. Paul Ammann, Sushil Jajodia, and Peng Liu. Recovery from malicious transactions. *IEEE Trans on Knowledge and Data Engineering*, 14:1167–1185, 2002.
2. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.
3. P. Chrysanthis. *ACTA, A framework for modeling and reasoning about extended transactions*. PhD thesis, University of Massachusetts, Amherst, Amherst, Massachusetts, 1991.
4. J. Eder and W. Liebhart. Workflow Recovery. In *Proceeding of Conference on Cooperative Information Systems*, pages 124–134, 1996.
5. M. M. Gore and R. K. Ghosh. Recovery in Distributed Extended Long-lived Transaction Models. In *In Proceedings of the 6th International Conference DataBase Systems for Advanced Applications*, pages 313–320, April 1999.
6. B. Kiepuszewski, R. Muhlberger, , and M. Orłowska. Flowback: Providing backward recovery for workflow systems. In *Proceeding of the ACM SIGMOD International Conference on Management of Data*, pages 555–557, 1998.
7. C. Lala and B. Panda. Evaluating damage from cyber attacks. *IEEE Transactions on Systems, Man and Cybernetics*, 31(4):300–310, July 2001.
8. Meng Yu, Peng Liu, and Wanyu Zang. Multi-Version Attack Recovery for Workflow Systems. In *19th Annual Computer Security Applications Conference*, pages 142–151, Dec 2003.