XLRF: A Cross-Layer Intrusion Recovery Framework for Damage Assessment and Recovery Plan Generation

Eunjung Yoon[1] and Peng Liu[2]

[1] Department of Computer Science and Engineering
Pennsylvania State University, PA, USA
eyoon@cse.psu.edu
[2] College of Information Sciences and Technology
Pennsylvania State University, PA, USA
pliu@ist.psu.edu

Abstract. Recovering mission-critical systems from intrusion is very challenging, where fast and accurate damage assessment and recovery is vital to ensure business continuity. Existing intrusion recovery approaches mostly focus on a single abstraction layer. OS level recovery cannot fully meet the correctness criteria defined by business process semantics, while business workflow level recovery usually results in *non-executable* recovery plans. In this paper, we propose a cross-layer recovery framework, called XRLF, for fast and effective post-intrusion diagnosis and recovery of compromised systems using the dependencies captured at different levels of abstraction; business workflow level and OS level. The goal of our approach is two-fold: first, to bridge the semantic gap between workflow-level and system-level recovery, thus enable comprehensive intrusion analysis and recovery; second, to automate damage assessment and recovery plan generation, thus expedite the recovery process, an otherwise time-consuming and error-prone task.

Keywords: cross-layer intrusion recovery, recovery plan, dependency graph, system calls.

1 Introduction

Intrusion, especially in mission-critical, enterprise systems, often results in the corruption of important data causing devastating effects to serious consequences such as significant financial loss. In fact, many mission critical systems have rather strict business continuity and availability requirements, and thus demand fast and efficient recovery from intrusion, which is essential for minimizing financial losses from cyber attacks.

Although a lot of effort has been devoted to the detection and prevention of malicious attacks, perfect prevention is still unobtainable. Intrusion detection can prevent the effects of the intrusion from spreading but cannot guarantee the integrity and availability of the compromised system. In some situations, an

S. Qing et al. (Eds.): ICICS 2013, LNCS 8233, pp. 194–212, 2013.

intrusion detection system (IDS) is also unable to discover all damage to the system and the damage can be spread without being detected by the IDS. We recognize that perfect intrusion detection is hard to achieve and some damage to the system even after the detection of the attack is always possible. To this end, we believe that an effective intrusion response and recovery scheme is essential for repairing compromised systems from the damage.

There has been growing interest in studies of intrusion recovery ([1], [8–10], [12], [14,15], [18], [20]), however most of intrusion recovery research has focused on a *single layer of abstraction*: Operating System (OS) level, Application level, or Business workflow level. None of them considered the problem of the *semantic gap* in infection diagnosis and recovery between high-level business workflow and the underlying infrastructure. Besides, there is a significant difference between *business workflow-level semantics* and *OS-level semantics*. Different abstraction layers may provide different granularity levels and different semantic views of the attack.

Due to the significant semantic gap between the two layers, an effective and comprehensive recovery may be very difficult. Most business-critical systems adopt the workflow management system with mission-critical processes [3] and thus, these systems, when intrusion detected either at workflow level or OS level, would require combined damage assessment and recovery by cooperation between the workflow layer and the underlying system layer.

In this paper, we present a *Cross-Layer Intrusion Recovery framework*, called XLRF, based on a combination of workflow-level and OS-level view. A cross-layer architecture allows us to have a global view about the intrusion and a more comprehensive understanding about recovery from the intrusion. Primary goal of our framework is to close the gap between business workflow-level and OS-level recovery semantics by focusing on two levels of abstraction in both, thus preserving workflow integrity, as well as system integrity. To the best of our knowledge, this is the first cross-layer recovery approach that focuses on both at business workflow level and OS level.

OS-level recovery approaches focus on low-level system events and do not take into account high-level business workflow implications. Due to the lack of higher workflow-level abstraction, OS-level intrusion recovery alone may cannot provide the comprehensive recovery solution, thus in many situations, needs manual work.

Similarly, *workflow-level recovery approaches* do not have enough information about low-level system activities that are very useful for fine grained intrusion analysis. Workflow-level recovery actions are generally performed at the granularity of business workflow tasks while OS-level recovery actions are generally performed at a granular file level, and thus workflow-level recovery approaches cannot handle the OS-level attacks (such as compromised processes, unauthorized data modification) properly. For example, workflow-level recovery performs task-level recovery actions (e.g., `undo` and `redo` of tasks), and thus task-level recovery actions cannot guarantee the removal of all the effects of the attack (compromised components) at the OS layer. As a result, workflow-level recovery

often results in *non-executable* recovery plan. In order for a recovery plan to be effective, it must be *executable* in reality, which consists of low-level system recovery operations. Therefore, neither the *OS-level recovery* nor the *workflow-level recovery* can provide a comprehensive and effective recovery solution.

Our cross-layer recovery framework explores the association between two different levels of abstraction by extracting information about the relationships between a business workflow and system call invocations from system call traces. We perform an automated analysis of system call log to semantically map OS-level dependencies to workflow-level dependencies. The damage assessment and recovery plan generation using dependency information is performed in both a *top-down* and *bottom-up* fashion between OS layer and workflow layer by exploiting the hierarchical relationship between the two layers.

Another goal of our recovery framework is to maximize automation in damage assessment and recovery plan generation. Traditionally, the recovery from an attack is manually performed by system administrators, which is time-consuming and error-prone. Automated response to intrusions has become a major issue in defending mission-critical systems, in which it is important to know how fast a problem can be resolved after it is detected, and distributed systems, in which manual diagnosis and repair is difficult. To this end, we propose automated recovery plan generation framework for fast recovery.

This paper makes the following contributions:

- We develop a cross-layer recovery framework that bridges the semantic gap between business workflow-level recovery and OS-level recovery. To the best of our knowledge, this work is the first to develop a cross-layer recovery framework that considers business workflow layer and OS layer.
- We provide automated damage assessment and generation of recovery plan that is *executable*.
- We point out the inherent problems with single layer intrusion recovery schemes.

The remainder of the paper is organized as follows. We describe related work in Section 2. Section 3 describes our running workflow example. Section 4 presents an overview of our cross-layer recovery framework, called XLRF. We present the details of XLRF design and implementation of three phases in Section 5, and the evaluation of XLRF in Section 6. In Section 7, we briefly revisit the limitations of single layer recovery Finally, we conclude the paper in Section 8.

2 Related Work

Existing intrusion analysis and recovery approaches focus either on the OS layer or the workflow layer (*single-layered approach*), whereas our work focuses on both the OS layer and the workflow layer (*cross-layered approach*).

OS-level Recovery. OS-level recovery approaches only focus on low-level system events and do not take into account high-level abstraction of workflow-level recovery, which is essential for most business and mission-critical systems.

ReVirt [5] focuses on intrusion analysis by using Virtual Machine logging and replay. ReVirt provides recovery capability using checkpoint and roll back, however, it removes both affected and legitimate changes. BackTracker [13] provides intrusion analysis tool of tracking the sources of an intrusion. BackTracker captures and uses system calls for analyzing problems on the process and file system level. BackTracker uses previously recorded system calls and constructs the dependency graph by using system call dependencies from the detection point and traces affected system events on files or processes. XLRF is closely related to BackTracker for computation of system call dependencies and the dependency graph generation. Taser [8] is an intrusion recovery system that determines the set of tainted file system-operations and reverts the tainted operations but preserves legitimate operations. Taser logs all process, file and network operations to identify the file system modification after intrusion and provides selective redo of legitimate file-system operations after an attack occurs. RETRO [12] analyzes OS-level system events to determine the source of an intrusion by recording action history graph. RETRO tries to minimize re-execution (selective redo) by predicates, refinement, and shephered re-execution. SHELF [18] is a self-recovery system that leverages the Virtual Machine Monitor and taint-analysis for dynamic dependency tracking and quarantine. SHELF logs system-level events to track the dependencies among the events, maintains the dependency graph, and quarantine the infected and malicious objects.

Workflow-level Recovery. As discussed in Section 1, workflow-level recovery approaches often result in *non-executable* recovery plan. Yu et al. [20] introduced theories and analytical experiments for on-line attack recovery of workflows. Their recovery system identifies all damages caused by the malicious tasks that are detected by an IDS and automatically repairs the damages based on data and control dependencies among workflow tasks. Our *workflow-level* damage assessment using workflow-level dependencies are based on this work. Eder et al. [6] introduced workflow recovery concepts for reliable and consistent execution of business processes in the presence of failures and excetions. They integrate workflow transactions into WFMSs so that processes are treated as workflow transactions and in the event of failures, a running process is aborted and compensated. However, this approach mainly focuses on workflow failure recovery, which is different from the intrusion recovery that removes the effects of intrusions.

Other Recovery Approaches. Polygraph [14] is a software layer that extends the functionality of weakly consistent replication systems to support compromise recovery. Polygraph is based on the replication technique and tries to recover from data corruption in weakly consistent replicated storage systems. Ammann et al. [1] presented the recovery approch to the problem of removing undesirable but committed transactions from databases. They detect the flow of contaminated transactions through a database and roll back those transactions that are affected directly or indirectly by contaminated transactions. Solitude [10] is an application-level isolation and recovery system that uses a

copy-on-write filesystem to limit attack propagation by sandboxing untrusted applications and providing an explicit file sharing.

3 Example

For our case study in this work, we develop a simplified version of *travel reservation system* running on the Apache web server. Our travel reservation system can be represented as a business workflow that consists of six tasks as follows.

T_1: Input travel information
T_2: Reserve a flight ticket and sign in or sign up
T_3: If the customer is signed as a member, reserve a hotel as a member.
T_4: If the customer is a member, apply any credit or promo code.
T_5: If the customer is a guest, reserve a hotel as a guest.
T_6: Make a payment

Figure 1 shows the workflow graph that represents our travel reservation system example. This model is based on [2] and [20]. The workflow has two choices of execution paths, P_1: $T_1T_2T_3T_4T_6$ and P_2: $T_1T_2T_5T_6$, but in each execution, only one path can be selected by T_2. An attack can change the execution path of the workflow. In this example, P_1 is the execution path led by an attack, and P_2 is the normal execution path without an attack.

The workflow shows control and data dependencies between tasks. For example, if task T_1 is a malicious task, tasks T_2 and T_4 would be affected by T_1 as they will read corrupted data from task T_1, and thus calculating wrong results as T_2 is *data dependent* on T_1 and T_4 is *data dependent* on T_2. Consequently, T_2 would make a wrong decision to execute tasks on path P_1, resulting in changing the normal execution path as T_3 and T_5 are *control dependent* on T_2.

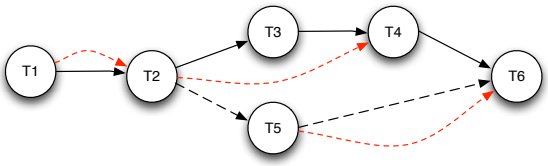

Fig. 1. Workflow of Travel Reservation System

4 XLRF Overview

We propose a *Cross-Layer Recovery framework*, called XLRF, an automated intrusion analysis and recovery plan generation framework. Our approach considers two different levels of abstraction: workflow level and OS level, and provides multi-level damage assessment and comprehensive recovery plan generation.

4.1 Assumptions

In this work, we assume the integrity of the system call log and checkpoints is preserved. We archive and send the system call logs collected on the web server to the trusted host that will investigate the effects of the intrusion and generate a recovery plan by using XLRF framework. We also assume that the administrator noticed an attack (i.e., system compromise) and identified at least one intrusion point before using XLRF, which is similar to the assumption of Backtracker [13] A sophisticated attacker may be able to successfully evade the IDS by manipulating a sequence of system calls (i.e., *mimicry attack* [17]). Recovery from mimicry or evasion attacks is out-of-scope for this paper, if these attacks have never been identified, as XLRF starts from the identified intrusion point.

4.2 Cross-Layer Recovery

Our cross-layer recovery framework (XLRF) is based on the analysis of workflow-level and OS-level control and data dependencies. We use the dependency information at each layer to analyze the effects of the intrusion and to automatically generate a recovery plan which consists of recovery actions that will revert the effects. XLRF takes inputs as a *workflow specification*, system call log recorded by the operating system, and the intrusion point identified by the administrator from IDS alerts. XLRF builds the association between workflow tasks and system calls based on the dependencies at the both layers. From the workflow-level perspective, each component at the OS layer has a corresponding task node at the workflow layer, and vice versa.

Damage assessment is performed in a combined *bottom-up* (from OS level to workflow level) and *top-down* (from workflow level to OS level) analysis. Once XLRF has completed damage assessment, it starts to generate a recovery plan with the result.

4.3 Workflow Level

A workflow typically represents a business process, which is composed of a sequence of tasks (i.e., business activities), and a set of dependencies that represent the relationships between the tasks.

Workflow Dependencies. The task dependencies in a business workflow is imperative to determine which tasks to recover and the order in which tasks are recovered. In this paper, we consider two types of dependencies between workflow tasks: *data dependency* and *control dependency.*

- *Data dependency.* Data dependencies (data flows) among tasks describe inputs and outputs of a task. Given a task T_i, we use $R(T_i, f)$ and $W(T_i, f)$ to denote a read operation and a write operation of task T_i on file f, respectively. Task T_j is data dependent on task T_i if $W(T_i, f)$ happens before $R(T_j, f)$ or $W(T_j, f)$.

  For example, if T_i modified file f and then T_j uses (reads or writes) the file for performing the task, task T_j is *data dependent* on task T_i. That is,

the state of T_j would depend on the value of the input file f as T_i has modified the file f that is shared by the two tasks. In Figure 1, dotted red lines represent data dependencies between tasks.

– *Control dependency.* The control dependency specifies the control flow of the workflow. Given two tasks T_i and T_j within a workflow, the task T_j is *control dependent* on T_i if T_j can be activated depending on the outcome of T_i. The output value of T_i determines the execution path of the workflow, either to execute T_j or another task. Thus, control dependencies of a workflow decide the execution order of tasks. In our running example shown in Figure 1, T_3 and T_5 are *control dependent* on T_2.

Workflow Dependency Graph (WDG). A workflow can be represented as a directed graph G(V, E), called *workflow dependency graph* (WDG), comprising V a set of nodes and E a set of directed edges, in which each node represents a task and directed edges represent dependencies between tasks, as shown in Figure 1. The WDG also can be generated by using workflow mining technique [16].

4.4 Operating System Level

XLRF uses OS-level information to identify system-level causal events (i.e., information flow) that connect OS objects, such as processes and files. The OS-level information can provide finer grained auditing and view of underlying system activities. For OS-level analysis, we are particularly interested in system calls and the data dependencies among system calls, which is similar to the approach that is generally used for behavioral malware detection in [4] and [19]. XLRF records all or selected system call invocations at run-time to extract system call dependencies from the system call traces.

System Call Dependencies. Every system call operation has a list of argument values and the return value, which we use for exploring *data dependencies* among system calls. We also use the timestamp of each system call to determine *the temporal order* of system calls. We extract dependencies among system calls for OS-level dependency analysis by analyzing each system call's arguments and the return value. In this paper, we only focus on *data dependencies* among system calls, which allow us to effectively identify data flows between system calls and to understand the semantics of the program and the effects of intrusion on system objects.

Christodorescu et al. [4] describe three types of dependencies among system calls: *def-use, ordering*, and *value* dependence but we only focus on data dependency (same as their *def-use* dependency) in this paper.

– *Data dependency*: Data dependencies between system calls can be computed by arguments (i.e., input) and a return value (i.e., output) of each system call. The return value of a system call can be used as the argument of subsequent system calls. For example, given two system calls sc_i and sc_j, system call sc_j is *data dependent* on sc_i if sc_j uses the valid return value of sc_i as its argument.

System Call Dependency Graph (SCDG). The data dependencies that we extracted from system call traces are represented as a directed graph G(V, E), called *system call dependency graph* (SCDG), such that nodes are system calls and edges represent data dependencies among system calls. We compute the dependencies by analyzing relationships between system call arguments and return values.

5 Design and Implementation

The XLRF framework provides automated damage assessment and recovery plan generation by analyzing dependencies within and across at the workflow layer and at the OS layer, and by identifying affected workflow tasks and system objects using the observed dependencies.

The XLRF framework consists of three main phases (Figure 2): dependency analysis, damage assessment, and recovery plan generation. During the dependency analysis phase, XLRF analyzes the system call log to determine dependencies among system calls and workflow tasks and constructs a *cross-layer dependency graph*, called XDG. In the damage assessment phase, XLRF identifies all the malicious or affected workflow tasks and system objects by traversing XDG from initially identified, malicious system objects (i.e., intrusion point). In the recovery plan generation phase, XLRF automatically generates a recovery plan based on the malicious or affected tasks and system objects that have been identified.

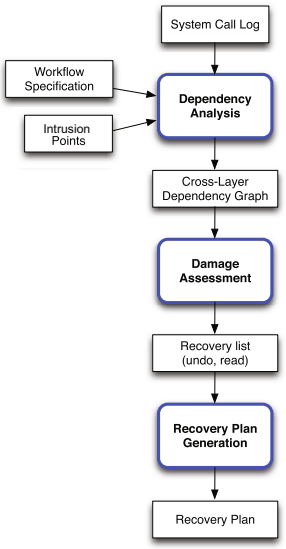

Fig. 2. The XLRF framework

5.1 Input

Workflow Specification. One of the inputs to XLRF is the *workflow specification*. The workflow specification contains the workflow type information that describes the workflow task structure (control flow) and information exchange between tasks (data flow) in a workflow. Each workflow is an instance of a workflow type. XLRF takes as input the workflow specification and system call traces to identify current running workflow instances.

System Call Log. The second input to XLRF is the system call log. System call traces are essential for XLRF to identify attack events and analyze dependencies between system calls. Logging mechanism will be discussed further in Section 5.3

Intrusion Point. Once an intrusion is detected by an IDS, the administrator can identify the intrusion point from the IDS alerts. XLRF takes as input the intrusion point and start to investigate the effects of the intrusion both at the OS layer and at the workflow layer to generate a recovery plan.

5.2 Output

Recovery Plan. The output of XLRF is a recovery plan that consists of a set of recovery actions. XLRF automatically generates a recovery plan which is very useful not only by reducing the system administrator's manual recovery process but also by minimizing human errors, and thus ultimately reducing the recovery time.

5.3 Logging

To build a proof-of-concept prototype of XLRF, we collect system call traces on the Apache HTTP server (`httpd`) using *DTrace* framework during normal execution of our *online travel reservation workflow* to track system events on OS-level objects such as processes, files and socket connections. *DTrace* is a dynamic tracing framework developed by Sun Microsystems. *DTrace* can dynamically instrument the running operating system kernel and running applications without rebooting the kernel or restarting applications. Gessiou et al. [7] also used *DTrace* framework for collecting data provenance information.

Each entry of the system call log contains detailed system object information such as process ID, file descriptor, and socket descriptor, the timestamp, and/or session ID of each system call invocation. During the dependency analysis phase, XLRF uses this information to construct a *system call dependency graph* (SCDG) and a *workflow dependency graph* (WDG). XLRF then constructs a *cross-layer dependency graph* (XDG) by associating the two graphs, SCDG and WDG; mapping of each node of SCDG (system call operation) and each node of WDG (workflow-level task).

XLRF can identify a user's workflow instance that corresponds to a particular system call and process. In our running example, once an authenticated session has been established, the session ID can be identified for all web page (`php` code file)

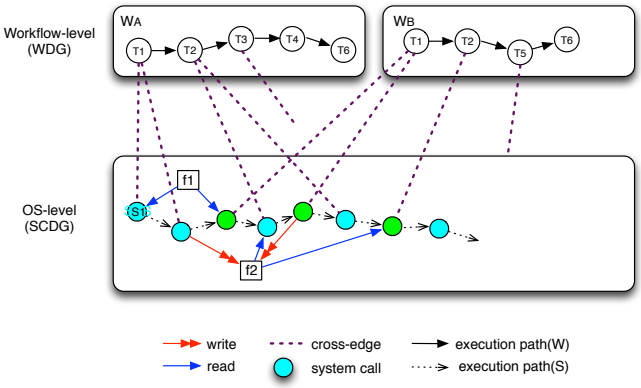

Fig. 3. Cross-Layer Depedency Graph

accesses. In addition, the entities involved in a workflow can be uniquely identified by a combination of a username and the workflow ID that is generated for every workflow of a user. This user identification allows us to observe whose workflow instance ID matches the current workflow instance (i.e., the origin of a system call and the corresponding workflow task), and thus enables selectively to recover only affected workflow instances. For example, in Figure 3, Alice's workflow instance (W_A) and Bob's workflow instance (W_B) at the workflow layer are mapped to each system call node at the OS layer based on our approach.

5.4 Dependency Analysis

Once an IDS has detected an intrusion, XLRF takes as input the intrusion point deduced by the administrator from the IDS alerts, the system call logs collected during normal execution, and the workflow specification. The system call log allows us to learn about the executed path of the workflow and dependencies among system calls. XLRF preprocesses the system call log by focusing particularly on file and process operations and extracts the data dependencies, and constructs *dependency graphs*. The dependency information is essential for our recovery framework to identify directly or indirectly affected objects from the intrusion which recovery is needed for. XLRF identifies current running workflow instances and their *workflow dependency graph* (WDG) with the workflow specification and the system call trace. Similarly, XLRF constructs the *system call dependency graph* (SCDG) with the dependencies among system calls extracted from the system call trace. XLRF then correlates and combines the two layers of dependency graphs as a single hierarchical graph, called *cross-layer dependency graph* (XDG), by associating semantic links between two graphs.

Cross-Layer Dependency Graph. *Cross-layer dependency graph* (XDG) enables cross-layer damage analysis, which allows us to identify the effects of

Table 1. Association of file information to workflow tasks

T_1	T_2	T_3	T_4	T_5	T_6
index.php (GET) flight.dat (R)	login.php (POST) guestinfo.php (POST) users.dat (R/W)	check.php (POST) users.dat (R)	member.php (GET) hotel.dat (R) promo.dat (R) invoice.dat (W)	guest.php (POST) hotel.dat(R) discount.php (POST) promo.dat (R) invoice.dat (W)	payment.php (POST) invoice.dat (R) credit.dat (R/W) payment.dat (W)

intrusion using dependencies collected both at the workflow layer and the OS layer.

After we constructed WDG and SCDG, we find the *semantic relationships (links)* between the two dependency graphs. Given WDG and SCDG, XLRF adds the edges that cross the two layers so that each *cross-edge* in the resulting XDG connects vertices of each graph at the two layers.

XLRF correlates the relationship between WDG and SCDG by analyzing the system call log and the workflow specification. Having a good criteria to associate each system call to a task of a particular workflow instance is challenging, as the system call log consists of multiple workflow instances involving multiple clients. XLRF separates the combined traces at the OS layer (SCDG) into separate *workflow instances* at the workflow layer (WDG) for each user. The *workflow specification* provides us information of interaction between a workflow task and its operations on some input and output data (e.g., data files). Therefore, XLRF takes both the workflow specification and system call logs as inputs and leverages the information to find out the semantic relationships between workflow-level and OS-level activities.

Table 1 shows the information obtained from the workflow specification used for our running workflow example. Our workflow example separates a workflow into a set of tasks with the semantic relationships between webpages, and data files specified in the workflow specification. Each workflow task basically comprises of program code files (`php` files) and data files. In this work, for the sake of simplicity, we use data files stored in file system instead of database. Using this information along with system call traces, XLRF can generate XDG, as shown in Figure 3, which is an integrated graph of WDG and SCDG, connected by the *cross-edges*.

5.5 Damage Assessment

During the damage assessment phase, XLRF identifies which tasks at the workflow layer and which system objects at the OS layer have been affected by malicious system objects. XLRF takes as inputs XDG that has been constructed in the dependency analysis phase, and the intrusion points (i.e., identified malicious objects) to create the *recovery list* of malicious or affected workflow tasks and system-level objects, as shown in Algorithm 1. This information is essential for deciding recovery actions that will be used for generating a recovery plan. The overview of recovery actions (`undo` and `redo`) will be given in the following section.

The analysis is performed on XDG, in both directions: *bottom-up* (SCDG→WDG) and *top-down* (WDG→SCDG):

Bottom-up Analysis. XLRF starts from the detection point to diagnose affected workflow tasks by using a bottom-up analysis.

At the OS layer (SCDG). Given the intrusion point, locate a malicious node of SCDG.

OS layer (SCDG) → Workflow layer (WDG). From the malicious node of SCDG, follow the *cross-edge* to locate the associated task node of WDG. Add the task to the *recovery list* (*workflow undo list*).

At the workflow layer (WDG). From the malicious task node, identify all affected tasks by using data and control dependencies (*dependency edges).* Add all affected tasks to the *recovery list* (*workflow undo list* or *workflow redo list*). For workflow-level analysis, we use the approach similar to Yu et al's workflow recovery [20] to determine which workflow tasks have been affected and need to be repaired.

We explain the analysis by using our workflow example and XDG, shown in Figure 1 and Figure 3. Suppose that file $f1$ has been identified as corrupted by an intrusion. From XDG shown in Figure 3, the first system call node s_1 of SCDG (`read f1`) will read the corrupted data (a). The system call node has a *cross-edge* to task node T_1 of Alice's workflow instance (W_A), and thus T_1 of WDG is marked as bad (malicious or affected) and added to Alice's workflow *undo list*, $WU_A = \{T_1\}$ because T_1 needs to be undone in the recovery (b). At the workflow layer (WDG), by the data dependencies shown in Figure 1, task T_2 is infected by task T_1 because it is *data dependent* on T_1, by reading a corrupted data ($f1$) from T_1 and then creating wrong results. Thus, T_2 is added to *undo list*, $WU_A = \{T_1, T_2\}$. Similarly, T_4 is *data dependent* on T_2, and thus added to the *undo list*, $WU_A = \{T_1, T_2, T_4\}$. Although task T_3 is not affected, it needs to be undone in case T_3 performed data write operations (`e.g., write f`), because the data will be corrupted if the execution path is changed by `redo`(T_2) and T_3 is not on the new execution path. XLRF adds T_3 to *candidate undo list*, $CU_A = \{T_3\}$. T_6 is also added to $CU_A = \{T_3, T_6\}$ because it will be *data dependent* on T_5 and get a wrong result from the current execution if the execution path is changed by `redo`(T_2). From the $WU_A = \{T_1, T_2, T_4\}$ and $CU_A = \{T_3, T_6\}$, XLRF creates *redo list* $WR_A = \{T_1, T_2, T_6\}$, because the tasks are *not control dependent* on any malicious or affected tasks. If any task is not on the new executing path (T_3, T_4), it does not need to be redone because it can create corrupted data (c).

Top-down Analysis. XLRF refines the *workflow-level recovery list* at the high level of abstraction (WDG) and bottoms out in a set of directly *executable OS-level recovery actions* at the low level of abstraction (SCDG) by using a top-down analysis. This analysis allows us to derive a recovery plan from recovery goals. XLRF can create the *OS-level recovery list* of affected system objects given malicious or affected tasks of a particular workflow instance, which will be used in the recovery plan generation phase.

Algorithm 1. Damage Assessment

Input:
XDG: Cross-layer dependency graph
M: Malicious Objects (from IDS alerts)
Output:
WU: Workflow Undo list
WR: Workflow Redo list
CU: Candidate Undo list
SU: System-call Undo list

1: **if** SCDG node s of XDG is data-dependent on M **then**
2: **if** cross-edge $e(s,t)$ exists **then**
3: Follow $e(s,t)$ and locate task node t
4: Add t to WU
5: **while** $succ(t)$ exists **do**
6: **for all** $succ(t)$ data-dependent on t **do**
7: Add $succ(t)$ to WU
8: $t \leftarrow succ(t)$
9: **if** $t_i \in succ(t)$, not data-dependent on t **then**
10: Add t_i to CU
11: **for each** WDG node t_i of XDG **do**
12: **if** t_i is control-dependent on t_j **then**
13: **if** $(t_j \notin WU)$ **or** $(t_j \in WU,\ t_i \in \text{succ}(\text{redo}(t_j)))$ **then**
14: Add t_i to WR
15: **for each** t_i of WU **do** ▷ backward
16: Follow cross-edge$e(s,t_i)$
17: **for all** node s belongs to t_i **do**
18: Add s to $SU[t_i]$

Workflow layer (WDG) → OS layer (SCDG). Given a malicious or affected task node of WDG, follow the *cross-edges* to identify the associated low-level system call nodes of SCDG.

At the OS layer (SCDG). From the identified system call nodes and dependencies among them, find all the affected system objects (files, in our example).

In our framework, redoing a workflow task will automatically re-execute all system calls associated with the task. Thus, XLRF only maintains *undo list* for system call-level operations (no *redo list*).

5.6 Recovery Plan Generation

The last phase of XLRF is to automatically generate a *recovery plan* based on the dependency information and the damage analysis results. Generated recovery plan describes *executable recovery actions* needed for reverting the effects of intrusion both at the workflow layer and at the OS layer.

Before discussing our recovery plan generation scheme, we briefly describe recovery actions.

Workflow Task Undo and Redo. To recover from intrusion, basically two operations: `undo` and `redo` are used. To remove all effects of intrusion, XLRF needs to `undo` malicious and affected tasks that have been identified during the damage assessment phase. XLRF creates *undo lists* for workflow tasks to revert all the effect and creates *redo lists* for tasks to restore legitimate but removed operations that have been affected by the attack.

System-Call Undo and Redo. As far as we know, there is no known way to actually `undo` the already executed system call. Alternatively, we could roll back the object (e.g., file) affected by the system call to the last checkpoint, which is commonly used for reverting `write` operation to a file, or we could ideally use an *inverse* operation if supported. Executing *inverse* operations can be substantially more efficient than checkpoint and rollback mechanism. A recent work [11] presents a new technique for *inverse* operations but their approach is

still limited for linked data structures, which needs to be extended if it is to be used in real systems.

In fact, XLRF generates a recovery plan for leveraging checkpoint and rollback mechanism for system call `undo` and removing system-call level `redo` because the task-level `redo` operation will automatically re-execute system calls that belong to the specific task. From OS-level recovery perspective, the task-level `redo` is too coarse-grained, resulting in some unnecessary system call `redo` operations can be included in a recovery plan. However, we do not focus on efficient selective-redo approach for system calls in this work. Nevertheless, our recovery plan generation framework can be easily adapted to the advanced recovery technique as needed.

Algorithm 2. Recovery Plan Generation

Input:
WU: Workflow Undo list
WR: Workflow Redo list
CU: Candidate Undo list
SU: System-call Undo list
Output:
P: Recovery Plan

1: **while** WU is not empty **do**
2: Get task T_i from WU ▷ backward
3: Add `undo`(T_i) to P
4: **while** $SU[T_i]$ is not empty **do**
5: Get system call s_i from $SU[T_i]$
6: **if** s_i is `write`(f) **then**
7: Add `rollback` (f) to P
8: **while** WR is not empty **do**
9: Get task T_i from WR ▷ forward
10: Add `redo`(T_i) to P
11: **if** redo(T_i) changes the execution path **then**
12: **while** CU is not empty **do**
13: Get task T_i from CU
14: **while** $SU[T_i]$ is not empty **do**
15: **if** s_i is `write`(f) **then**
16: **if** s_i is the first `write` **then**
17: Add `undo`(T_i) to P
18: Add `rollback` (f) to P

The Order of Recovery Actions. To preserve correctness during the repair, the order of recovery actions needs to be correctly determined. The following rules describe the correct order of recovery actions from a workflow perspective.

- `undo` actions are performed in reverse order.
- `redo` actions are performed by following the original order of the task operations.
- For any task , its `undo` action should be done before its `redo` action.
- For any two tasks that modify the same file in order, the later task should be undone first before the earlier task is redone. For example, suppose that both task t_i and task t_j modify the same file f, $W(t_i, f)$ precedes $W(t_j, f)$. In this case, `undo`(t_j) should be done before `redo`(t_i).

XLRF automatically generates a recovery plan by the rules with *undo list* and *redo list* obtained during the damage assessment phase (see Algorithm 2).

5.7 Implementation

We implemented a proof-of-concept prototype of the XLRF framework on Linux based on the detailed design and algorithms that we have presented. We developed a simple web-based travel planning service in *PHP* running on Apache web

server as our running example. Our implementation does not need to make any changes to existing software components.

For logging system-level activities, we use *DTrace* framework to record selected system call invocations (e.g., read(), write()) on `httpd` process. While we do not focus on performance degradation problem in this work, logging overhead using *DTrace* was not a big concern, as *DTrace* has been designed to operate with low overheads when enabled, and zero or near-zero overhead when not enabled (selective instrumentation). We store the logs collected from the web server on a trusted platform that is isolated from the web server and we run the XLRF framework on the trusted platform so it does not incur much overhead to the HTTP server. We do not invent the wheel to prevent the integrity of the XLRF framework in this work but it is also hard to compromise the XLRF framework by the intrusion on the web server.

We implemented XLRF in *Perl* and *XML* for log analysis and automated recovery plan generation. We selected *Perl* script language as *Perl* is powerful for regular expressions processing. We generate dependency graphs using the dependencies obtained in the dependency analysis phase. The dependency graphs can be visualized using *Graphviz* as desired. XRLF then generates a recovery plan as an *XML* file that is human-readable and machine-readable. *XML* provides a basic syntax that can be used for sharing information between different platforms and applications. Manual or automated execution of a recovery plan using scripts is also much easier with *XML*. *Perl* also provides the features of *XML parsing* and converting it to *Perl* data structures.

6 Evaluation of Recovery Plan

We evaluated the correctness of our *cross-layer recovery framework* using several intrusion scenarios. In this paper, we present the evaluation of two scenarios due to space limit. We ran XLRF for each scenario on a trusted platform and compared generated recovery plans with the manually derived dependencies and expected recovery actions. We argue that XLF correctly generates a recovery plan for each attack scenario based on the evaluation.

Scenario 1: Data File Compromise

In our running example, an attacker can modifty the content of an invoice file (*invoice.dat*), in order to intentionally change the price of a particular travel plan, for example, from $2000 to $1000. In this scenario, Alice's workflow tasks T_4 and T_6 will be affected by the compromise, and thus will lead to a financial loss.

Recovery Goals. Revert *invoice.dat* and all affected files by rolling back the file to the last checkpoint to remove the effect from compromise. Remove all the effects and restore operations (`undo` and `redo`).

1. ***Generated Recovery Plan: (for Alice)***

⟨plan name= "rW_A"⟩
⟨action="uT_6"⟩ undo(T_6)
⟨subaction="T_6w11"⟩
rollback($payment.dat$)
⟨/subaction⟩
⟨/action⟩
⟨action="uT_4"⟩ undo(T_4)
⟨subaction="T_4w12"⟩
rollback($invoice.dat$)
⟨/subaction⟩
⟨ /action⟩
⟨action="rT_4"⟩ redo(T_4) ⟨/$action$⟩
⟨action="rT_6"⟩ redo(T_6) ⟨/$action$⟩
⟨/plan⟩

2. ***Derived Dependencies (manual):***
by Attacker:
write (invoice.dat, $badInput$)→**invoice.dat′**
by Alice's workflow: from T_6
read(payment.php)→read(invoice.dat′)→ read(credit.dat)
→write(payment.dat′)
Workflow level:
undo(T6) → undo(T4) → redo(T4) → redo(T6)
OS level (system call):
* Need undo? (Y/N)
T_6: w(payment.dat):Y ⇒ rollback (payment.dat)
T_6: r(credit.dat),r(invoice.dat), r(payment.php): N
T_4: w(invoice.dat):Y ⇒ rollback(invoice.dat)
T_4: r(promo.dat), r(hotel.dat), r(member.php): N
* Need redo? (Y/N)
T_4: r(member.php), r(hotel.dat), w(invoice.dat): Y

System-call level redo actions are automatically performed by their task-level redo, thus do not need to be added to a recovery plan.

Scenario 2: Execution Path Change. An attacker modifies *login.php* file and changes the execution path, allowing a guest member to be redirected to the webpage that only registered member can access. All guest members can benefit from this attack by making her travel plan using a member-only promotion, but resulting in a financial loss to the travel agency (*workflow violation*). Suppose that Figure 3 shows Alice (guest member)'s new execution path after the attack. Her original execution is $T_1 \rightarrow T_2 \rightarrow T_5 \rightarrow T6$, but it has been changed to $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_6$, respectively.

Recovery Goals. Revirt all malicious (*index.php*) or affected files and operations from the attack and restore original execution path.

1. ***Generated Recovery Plan: (for Alice)***

⟨plan name= "rW_A"⟩
⟨action="uT_6"⟩ undo(T_6)
⟨subaction="T_6w11"⟩
rollback($payment.dat$)
⟨/subaction⟩
⟨/action⟩
⟨action="uT_4"⟩ undo(T_4)
⟨subaction="T_4w12"⟩
rollback($invoice.dat$)
⟨/subaction⟩
⟨action="uT_2"⟩ undo(T_2)
⟨subaction="T_2w11"⟩
rollback($users.dat$)
⟨/subaction⟩
⟨ /action⟩
⟨action="rT_2"⟩ redo(T_2) ⟨/$action$⟩
⟨action="rT_6"⟩ redo(T_6) ⟨/$action$⟩
⟨/plan⟩

2. ***Derived Dependencies (manual):***
by Attacker:
write (users.dat, $badInput$) → **users.dat′**
by Alice's workflow: from T_2
read(login.php)→read(guestinfo.php)→ write(users.dat′)
→read(check.php)→ read(users.dat′) →read(member.php)→
read(hotel.dat)→read(promo.dat)→ write(invoice.dat′)
→read(payment.php) →read(invoice.dat′)→read(credit.dat)
→ write(payment.dat)
Workflow level:
undo(T6) → undo(T4) → undo(T2) → redo(T2) →
redo(T6)
OS level (system call):
* Need undo? (Y/N)
T_6: w(payment.dat):Y ⇒ rollback (payment.dat)
T_6: r(credit.dat), r(invoice.dat), r(payment.php): N
T_4: w(invoice.dat):Y ⇒ rollback(invoice.dat)
T_4: r(promo.dat), r(hotel.dat), r(member.php): N
T_2: w(users.dat):Y ⇒ rollback (users.dat)
T_2: r(guestinfo.php), r(login.php): N
* Need redo? (Y/N)
T_2: r(login.php), r(guestinfo.php), r(users.dat): Y
T_6: r(payment.php), r(invoice.dat), r(credit.dat), w (payment.dat): Y

Task T_4 is not on the re-execution path, thus T_4 needs not to be redone. The generated recovery plan shows for each user after rollback(*login.php*) has been done. The file *login.php* is shared by all users, so need to recover separately, before recover any workflow instance. The comparison shows that the recovery plan for Scenario 2 generated by XLRF is correct.

In all the scenarios mentioned above, we show that our approach is effective in damage assessment and recovery plan generation for intrusion recovery.

7 Revisiting the Limitations of Single Layer Recovery

As we discussed earlier, single layer recovery approaches cannot provide the comprehensive damage assessment and recovery solution due to the semantic gap between the high-level workflow abstraction and the low-level OS-level abstraction. Here we revisit and discuss the limitations of single layer recovery approaches: workflow-level and OS-level recovery with our running example.

A host-based IDS can monitor system activities so the administrator can identify the intrusion point such as a corrupted file, from the IDS alerts, which is used for OS-level recovery. By using data dependencies among system calls, all the affected files can be identified and recovered using the checkpoint and roll-back scheme and some of system call `redo` operations. However, even after all the corrupted files are recovered at the OS layer, the recovery still cannot ensure the correctness in business workflow semantics.

Let's revisit Scenario 1, OS-level recovery will do: `rollback`(payment.dat), `rollback`(invoice.dat), and `redo` operations on the files, such as `read` (invoice.dat), and `write`(payment.dat), however at the end of this OS-level recovery, only part of task T_6 has been recovered; `read`(payment.php) of T_6 will not be redone. From a workflow perspective, the client's payment process needs to be cancelled and re-executed, thus the entire task needs to be undone and redone for the client's input (She can probably change her mind later due to the price increase), so we need to redo the entire task T_6. Without the association between OS-level semantics and workflow-level semantics, the identification of current workflow instances that are affected and need to be repaired, will be very challenging. Therefore, we argue that OS-level recovery approach cannot provide correct recovery actions for high-level business workflow as it cannot determine the damage in the business workflow semantics correctly.

Workflow-level recovery scheme does not have the semantic information about the low-level system activities such as system call invocations. Therefore, in Scenario 1, when *invoice.dat* file has been compromised, workflow-level recovery does not aware about the intrusion and the system-level damage until any anomalous task of a workflow (e.g., task abortion) has been detected. It could never been detected in case of normal execution of the task even with a malicious data. Workflow-level approach provides task-level recovery so may cannot perform fine-grained recovery actions such as single file operation. Most workflow-level recovery approaches use workflow-level checkpointing resulting in expensive coarse-grained recovery. Therefore, workflow recovery often results in *non-executable* recovery plan as it cannot perform recovery actions at the OS layer, which requires the system administrator's manual process.

8 Conclusion

In this paper, we have first presented a cross-layer recovery framework for automatically analyzing the damage caused by intrusion and generating a recovery plan. We addressed the problem of single layer recovery approaches and proposed a new cross-layer recovery approach that takes into account both business workflow-level and OS-level recovery for providing a comprehensive recovery. We developed a proof-of-concept prototype of our recovery framework, called XLRF, that comprises dependency analysis, damage assessment, and recovery plan generation phases. We evaluated the effectiveness of our cross-layer recovery framework with several attack scenarios. XLRF correctly identified the effects of the intrusion and generated recovery plans for reverting all the effects from intrusion both at the workflow layer and at the OS layer.

Acknowledgments. This work was supported by ARO W911NF-09-1-0525 (MURI), AFOSR FA9550-07-1-0527 (MURI), NSF CNS-0905131, and AFOSR W911NF1210055.

References

1. Ammann, P., Jajodia, S., Liu, P.: Recovery from malicious transactions. IEEE Trans. on Knowl. and Data Eng. 14(5), 1167–1185 (2002)
2. Atluri, V., Ae Chun, S., Mazzoleni, P.: Chinese wall security for decentralized workflow management systems. J. Comput. Secur. 12(6), 799–840 (2004)
3. Balzarotti, D., Cova, M., Felmetsger, V.V., Vigna, G.: Multi-module vulnerability analysis of web-based applications. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS 2007, pp. 25–35. ACM, New York (2007)
4. Christodorescu, M., Jha, S., Kruegel, C.: Mining specifications of malicious behavior. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC-FSE 2007, pp. 5–14. ACM, New York (2007)
5. Dunlap, G.W., King, S.T., Cinar, S., Basrai, M.A., Chen, P.M.: Revirt: enabling intrusion analysis through virtual-machine logging and replay. SIGOPS Oper. Syst. Rev. 36(SI), 211–224 (2002)
6. Eder, J., Liebhart, W.: Workflow recovery. In: Proceedings of the First IFCIS International Conference on Cooperative Information Systems, COOPIS 1996, pp. 124–134. IEEE Computer Society, Washington, DC (1996)
7. Gessiou, E., Pappas, V., Athanasopoulos, E., Keromytis, A.D., Ioannidis, S.: Towards a universal data provenance framework using dynamic instrumentation. In: Gritzalis, D., Furnell, S., Theoharidou, M. (eds.) SEC 2012. IFIP AICT, vol. 376, pp. 103–114. Springer, Heidelberg (2012)
8. Goel, A., Po, K., Farhadi, K., Li, Z., de Lara, E.: The taser intrusion recovery system. In: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, SOSP 2005, pp. 163–176. ACM, New York (2005)
9. Hsu, F., Chen, H., Ristenpart, T., Li, J., Su, Z.: Back to the future: A framework for automatic malware removal and system repair. In: Proceedings of the 22nd Annual Computer Security Applications Conference, ACSAC 2006, pp. 257–268. IEEE Computer Society, Washington, DC (2006)

10. Jain, S., Shafique, F., Djeric, V., Goel, A.: Application-level isolation and recovery with solitude. In: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008, Eurosys 2008, pp. 95–107. ACM, New York (2008)
11. Kim, D., Rinard, M.C.: Verification of semantic commutativity conditions and inverse operations on linked data structures. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 528–541. ACM, New York (2011)
12. Kim, T., Wang, X., Zeldovich, N., Kaashoek, M.F.: Intrusion recovery using selective re-execution. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI 2010, pp. 1–9. USENIX Association, Berkeley (2010)
13. King, S.T., Chen, P.M.: Backtracking intrusions. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP 2003, pp. 223–236. ACM, New York (2003)
14. Mahajan, P., Kotla, R., Marshall, C.C., Ramasubramanian, V., Rodeheffer, T.L., Terry, D.B., Wobber, T.: Effective and efficient compromise recovery for weakly consistent replication. In: Proceedings of the 4th ACM European Conference on Computer Systems, EuroSys 2009, pp. 131–144. ACM, New York (2009)
15. Paleari, R., Martignoni, L., Passerini, E., Davidson, D., Fredrikson, M., Giffin, J., Jha, S.: Automatic generation of remediation procedures for malware infections. In: Proceedings of the 19th USENIX Conference on Security, USENIX Security 2010, p. 27. USENIX Association, Berkeley (2010)
16. van der Aalst, W., Weijters, T., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE Trans. on Knowl. and Data Eng. 16(9), 1128–1142 (2004)
17. Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS 2002, pp. 255–264. ACM, New York (2002)
18. Xiong, X., Jia, X., Liu, P.: Shelf: Preserving business continuity and availability in an intrusion recovery system. In: Proceedings of the 2009 Annual Computer Security Applications Conference, ACSAC 2009, pp. 484–493. IEEE Computer Society, Washington, DC (2009)
19. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: capturing system-wide information flow for malware detection and analysis. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS 2007, pp. 116–127. ACM, New York (2007)
20. Yu, M., Liu, P., Zang, W.: Self-healing workflow systems under attacks. In: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS 2004), pp. 418–4025. IEEE Computer Society, Washington, DC (2004)