Automatic intrusion recovery with system-wide history

by

Taesoo Kim

B.S., Korea Advanced Institute of Science and Technology (2009)
S.M., Massachusetts Institute of Technology (2011)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2014

Signature redacted

Author .....
Department of Electrical Engineering and Computer Science
May 18, 2014

Signature redacted

Certified by .....
Nickolai Zeldovich
Associate Professor
Thesis Supervisor

Signature redacted

Accepted by .....
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Students

Automatic intrusion recovery with system-wide history

by

Taesoo Kim

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2014, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Compromises of our computer systems are inevitable. New software vulnerabilities are discovered and exploited daily, but even if the software is bug-free, administrators may inadvertently make mistakes in configuring permissions, or unaware users may click on buttons in application installers with little understanding of its consequences. Unfortunately, recovering from those inevitable compromises leads to days and weeks of wasted effort by users or system administrators, with no conclusive guarantee that all traces of the attack have been cleaned up.

This dissertation presents RETRO, an automatic recovery system that repairs a computer after an adversary compromises it, by undoing the adversary's changes while preserving legitimate user actions, with minimal user involvement. During normal operation, RETRO records an *action history graph* to describe the system's execution, enabling RETRO to trace the adversary's changes and their effects. During repair, RETRO uses the action history graph to undo an unwanted action and its indirect effects by first rolling back its direct effects, and then re-executing legitimate actions that were influenced by that change. To minimize re-execution and user involvement, RETRO uses *predicates* to *selectively re-execute* only actions that were semantically affected by the adversary's changes, uses *refinement* to represent high level semantics into the action history graph, and uses *compensating actions* to handle external effects.

An evaluation of a prototype of RETRO for Linux with 2 real-world attacks, 2 synthesized challenge attacks, and 6 attacks from previous work, shows that RETRO can handle a wide range of real attacks with minimal user involvement, and preserve user's changes by efficiently re-executing parts of an action history graph. These benefits come at the cost of 35–127% in execution time overhead and of 4–150 GB of log space per day, depending on the workload. For example, a HotCRP paper submission web site incurs 35% slowdown and generates 4 GB of logs per day under the workload from 30 minutes prior to the SOSP 2007 deadline. We believe those overheads are acceptable in systems whose integrity is critical in their operations.

Thesis Supervisor: Nickolai Zeldovich
Title: Associate Professor

Acknowledgments

I would like to thank Nickolai Zeldovich, Frans Kaashoek and Robert Morris, for their invaluable comments and endless supports, in the course of my entire graduate study. Special thanks to Ramesh Chandra, for being a great collaborator, dear friend, and wholehearted mentor, ever since I met him in my first year at MIT. I also owe a lot to my colleagues in PDOS, namely, Silas, Haogang, Austin, Yandong, Neha, Meelap, Xi, Cody, Alex, Raluca, Jonas, Albert, and Frank, who enlightened me every day.

Thanks my parents, Junghee and Younghye, my sister Munjung, and parents-in-law, Jonghae and Sooyoung, for their endless love and support. Without their patience, I could not have achieved my child dream of being a scientist. I also thank my love, Michelle, and our baby, Issac, for always being there. I cannot imagine a single day of my life without their love and smile.

Last but not least, I thank my God for everything. My life always pursues of being a follower of Jesus, good son, beloved husband, and now proud father.

Most of the work in this thesis was published as "Intrusion Recovery Using Selective Re-execution" in *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI 2010)*.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

On October 3rd 2011, kernel.org [21] was back online after being offline for more than a month—it took that long for the site's administrators (who are skilled security professionals) to recover its integrity and functionality from an attack that was detected on August 28th. The administrators noticed the break-in 17 days after it happened, during which time the attack propagated to several servers. After taking a full month to investigate the attack, the only recovery option that the administrators had was to re-install all suspect servers and restore data from the latest backup taken before the attack, which resulted in loss of all data since then.

This example illustrates the challenges in performing intrusion recovery with state-of-the-art techniques: not only are they manual, tedious, and time-consuming, but they also lose legitimate changes made by users after recovery. The example also shows the importance of intrusion recovery: even a system with up-to-date security defenses is likely to get compromised at some point of time, as the system's overall security is only as strong as its weakest component. A single bug or misconfiguration in the system can result in a compromise, and when that happens, we need to recover the system's integrity to continue to run the system.

Unfortunately, it is hard to perform recovery correctly. Since many adversaries go to great lengths to prevent the compromise from being discovered, it can take days or weeks for users to discover that their machine has been broken into, making it hard to distinguish modifications made by adversaries from legitimate changes that have

been done by users. For example, if a user and an adversary modified the same file, we first need to distinguish the adversary's changes from the legitimate user's, and then to remove just the adversary's modifications for recovery. Moreover, even after recovery, it is hard to guarantee that the attack is completely removed from the computer system, because administrators recover from the attack based on incomplete knowledge of what were the attacker's effects and how the attack penetrated into the system. For example, even after recovery, it is possible that the system still has a backdoor installed by an attacker or a hidden account that the attacker might want to use later.

This dissertation presents the design and implementation of RETRO, a recovery system that automatically repairs a computer after an adversary compromises it, by undoing the adversary's changes while preserving legitimate user changes. With RETRO, users can remove all causal effects of an attack, by producing a system state as if the attack never occurred in the past. Our evaluation shows that RETRO can automatically recover from a wide range of real attacks, and its repair is several orders of magnitude faster than the original execution, especially for attacks that affect only a small part of the system's state. These benefits come at the cost of 35–127% in execution time overhead and of 4–150 GB of log space per day, depending on the workload. We believe those overheads are acceptable in systems whose integrity is critical in their operations.

1.1 Challenges

To provide automatic recovery in RETRO, we need to address three key challenges. First, the effects of an attack could be entangled with the user's legitimate changes. For example, a user's application may read or write to a file that was modified by an attacker (e.g., an admin adds a user's account to `/etc/passwd` after an attacker added his own account). To ensure complete repair, an ideal recovery system must be able to revert all causal effects of the offending actions, but preserve changes made by legitimate users (e.g., after recovery, we should remove the attacker's account but preserver the user's account added after the intrusion).

Second, preserving legitimate changes is non-trivial; there are lots of challenging

situations that can arise during recovery. For example, if a legitimate user modified the same line of a file that an adversary modified (e.g., an admin modified a user's home directory field in `/etc/passwd` that an attacker created), it is hard to decide what modification made by the user should be preserved (e.g., should we remove the entire attacker's account, including the home directory modified by an admin?). During recovery, if an administrator meets a situation where the context of the legitimate user's actions in the past does not match to the context in the new state, then we call this situation *conflicted*—we cannot figure out how to replay the user's past actions in the new repaired state during recovery. This potentially conflicting situation can be resolved by asking users how to correctly recover from this situation (e.g., asking what to preserve in `/etc/passwd`). An ideal system would require user's assistance only if it is not possible to resolve automatically, and perform all other parts of the repair unassisted.

Lastly, recovering from attacks should be fast, although attacks are detected days or weeks after the initial break-ins. To efficiently recover from these intrusions, the performance of recovery should not depend on the system's past execution time, but rather depend on the amount of effects that an attack introduced: if an attack affects small parts of the system's state, then its recovery should be quick by repairing those parts.

1.2 Approach

The core idea of RETRO's approach to automatic recovery is to keep track of a system-wide history that represents details of the system's execution. This execution history enables RETRO to correctly distinguish the effects of an attack from legitimate user's changes, and ultimately recover from the attack. To recover from an attack, RETRO performs recovery using *rollback-and-replay*: performing *rollback* to remove attacker's effects, and performing *replay* to preserve legitimate user's changes. Conceptually, once an attack is identified by an administrator, RETRO rolls back the state of the system to a time before the attack (to undo the attack's effects), skip the attacker's actions (not to reproduce attacker's changes), and re-executes all subsequent actions (to reapply legitimate changes).

With rollback-and-replay, RETRO can remove attacker's effects simply by rolling back the system's state to an earlier state that is known to be safe. To reconstruct changes, RETRO replays all the actions performed by legitimate users (without the attacker's action). However, this naive form of rollback-and-replay is not practical to repair an attack happened a long time ago: if an attack was detected a month after the break-in, then the entire system should roll back to a safe backup taken a month ago, and should replay a month of users actions performed after the initial break-in.

To make recovery efficient, RETRO uses a fine-grained rollback-and-replay technique that uses a detailed dependency graph of the system's execution. RETRO can first roll back only the attacker's changes and re-execute a part of the dependency graph for recovery. Furthermore, RETRO avoids unnecessary re-execution of actions whose inputs are semantically identical to their inputs during original execution, by comparing effects of actions in both executions during replay.

To make the recovery automatic, RETRO attempts to minimize the user's involvement during replay. RETRO represents precise dependencies at multiple levels of abstractions, giving rich semantics to the dependency graph, so RETRO can infer the user's intention precisely with the rich context recorded in the past. Whenever RETRO identifies conflicts that cannot be resolved automatically, it notifies users (or asks about their decision) to resolve these conflicts (or to handle external effects) during repair.

With those techniques, RETRO can disentangle the adversary's changes from the current system's state, and repair the computer system as if the intrusion never happened. Our approach works in well-contained environments where all dependencies are completely captured and actions in the action history graph are faithfully replayable—when an action is replayed, its result is semantically identical to the result that was recorded in the past. However, for systems where it is hard to capture complete dependencies, like web applications (e.g., missing a dependency whenever a user interacts with), RETRO's approach requires more elaborate mechanisms to capture those dependencies to correctly recover from attacks. We discuss new research projects aiming to make RETRO's idea applicable to a variety of those environments, including web applications in §1.6.

1.3 Techniques

How can RETRO disentangle unwanted actions from legitimate operations, and undo all effects of the adversary's actions that happened in the past, while preserving every legitimate action? RETRO addresses these challenges with five ideas:

First, RETRO models the entire system using a new form of a dependency graph, which we call an *action history graph*. Like any dependency graph, the action history graph represents objects in the system (such as files and processes), and the dependencies between these objects (corresponding to actions like reading a file by a process). Each dependency in the action history graph captures its semantics (e.g., the arguments and return values of an action).

To record precise dependencies, the action history graph supports *refinement*, that is, representing the same object or action at multiple levels of abstraction. For example, a directory inode can be refined to expose individual file names in that directory, and a process can be refined into function calls and system calls.

Second, RETRO *re-executes* actions in the graph, such as system calls or process invocations, that were influenced by the offending changes. For example, undoing undesirable actions may indirectly change the inputs of later actions, and thus these actions must be re-executed with their repaired inputs.

Third, RETRO *selectively* re-executes those actions in the graph, to make recovery fast. For example, to efficiently recover from an attack that affects small parts of the system's state, RETRO first rolls back only affected parts of the system, and initiate re-execution of actions that might be affected by these rollbacks, by analyzing action history graph.

Finally, during selective re-execution, RETRO uses *predicates* to identify actions whose dependencies are semantically different after repair, thereby minimizing cascading re-execution. For example, if an attacker modified some file, and that file was later read by process P, we may be able to avoid re-executing P if the part of the file accessed by P is the same before and after repair.

1.4 Assumptions

RETRO makes two important assumptions to correctly recover from attacks.

First, we consider the underlying systems, such as the kernel and RETRO itself, as our trusted computing base, meaning that logs of RETRO or the kernel cannot be tampered with or compromised. In RETRO, we rely on existing techniques for hardening the kernel, such as [19, 33, 45, 47], to achieve this goal in practice.

Second, RETRO assumes that the user promptly detects any intrusions with wide-ranging effects (or abnormal behavior) on the system. If such intrusions persist for a long time without being noticed by users, RETRO will require re-execution of large parts of the system, and potentially incur many conflicts with significant user input required. However, we believe this assumption is reasonable in practice, since the goal of many adversaries is to remain hidden for as long as possible, in order to maximize their incentive of controlling the machine (e.g., by sending more spam, or running as a botnet).

1.5 Summary of results

Using a prototype of RETRO for Linux, we show that RETRO can recover from both real-world and synthetic attacks, while preserving legitimate user changes. Out of ten experiment scenarios, six required no user input to repair, two required user confirmation that a conflicting login session belonged to the attacker, and two required the user to manually redo affected operations. We also show that RETRO's ideas of refinement and predicates are key to repairing precisely the files affected by the attack, and to minimizing user involvement. A performance evaluation shows that, for extreme workloads that issue many system calls (such as continuously recompiling the Linux kernel), RETRO imposes a 89–127% runtime overhead and requires 100–150 GB of log space per day. For a more realistic application, such as a HotCRP [28] conference submission site, these costs are 35% and 4 GB per day, respectively. RETRO's runtime cost can be reduced by using additional cores, amounting to 0% for HotCRP when one core is dedicated to RETRO.

1.6 Research impact

The idea of the history-based record-and-replay approach that RETRO proposed as a recovery mechanism have inspired four new systems: WARP [11] and AIRE [12] for automatic recovery, and POIROT [25] and RAIL [14] for auditing past intrusions.

Although we believe the recovery approach of RETRO is well-suited for a server or workstation environment, RETRO's automatic repair scheme is not a good fit for systems that have lots of interaction to users or incur lots of external dependencies, which cannot be captured in a machine running RETRO: for example, web applications (lots of interactions with users) or distributed services (missing external dependencies between web services) are not well-suited models for RETRO's approach. However, by providing a way to resolve such external or missing dependencies, the idea of RETRO can easily be extended to support these environments; we extended its approach to WARP for web applications and to AIRE for distributed web services.

WARP applies the idea of RETRO to the context of web applications, by reconnecting external dependencies between web services and browsers, combining with browser-level record-and-replay techniques. WARP introduces a variety of new techniques such as retroactive patching and a time-travel database, built on the conceptual foundation of RETRO.

AIRE extends our idea further to distributed web services. Unlike RETRO and WARP, additional external dependencies (i.e., requests between servers) can be resolved in AIRE, as long as web servers are running AIRE or are exposing a set of predefined APIs that AIRE proposed. Therefore, even if an attack propagates and affects the state of other systems, AIRE can correctly identify and recover from the attack, by communicating each others.

While building these recovery systems, we realized that detecting intrusions is a prerequisite for recovery and that it is an important, non-trivial problem in itself. This was the motivation for POIROT [25], a system that efficiently audits all past requests to a web application for attacks that exploited a vulnerability, using just a patch fixing the vulnerability.

RAIL extends our idea to an auditing system in the context of a web framework. Since RAIL implements the history-based record-and-replay on top of a web framework, it can capture rich semantics from applications (e.g., Meteor, a web framework RAIL uses, has a clear separation between browsers and server, and clearly identifies which data propagate from server to browsers). Given these rich semantics, RAIL could realize RETRO's idea as an efficient auditing system for web applications.

In summary, this dissertation opened new possibilities for building systems that are easy to recover: given a patch for a new security vulnerability, we can automatically identify all past intrusions that exploited this vulnerability, undo all the changes made by these intrusions, and preserve all legitimate user changes. The dissertation, in particular, focuses on the original idea of an automatic recovery mechanism by using a history-based record-and-replay technique.

1.7 Organization

The rest of the dissertation is organized as follows. §2 presents an overview of RETRO's architecture and workflow. §3 discusses RETRO's action history graph in detail, and §4 describes RETRO's repair managers. Our prototype implementation is described in §5, and §6 evaluates the effectiveness and performance of RETRO. Finally, §7 discusses the limitations and future work, and §9 concludes the thesis of this dissertation.

Chapter 2

Overview

RETRO consists of several components, as shown in Figure 2-1. During normal execution, RETRO's kernel module records a log of system execution, and creates periodic checkpoints of file system state. When the system administrator notices a problem, he or she uses RETRO to track down the initial intrusion point. Given an intrusion point, RETRO reverts the intrusion, and repairs the rest of the system state, relying on the system administrator to resolve any conflicts (e.g., both the adversary and a legitimate user modified the same line of the password file). The rest of this section describes these phases of operation in more detail with a running example, and outlines the assumptions made by RETRO about the system and the adversary.

2.1 Running example

To understand challenges facing RETRO, let's consider the following attack, which we will use as a running example. Eve, an evil adversary, compromises a Linux machine, and obtains a root shell. To mask her trail, she removes the last hour's entries from the system log. She then creates several backdoors into the system, including a new account for `eve`, and a PHP script that allows her to execute arbitrary commands via HTTP. Eve then uses one of these backdoors to download and install a botnet client. To ensure continued control of the machine, Eve adds a line to the `/usr/bin/texi2pdf` shell script (a wrapper for LaTeX) to restart her bot. In the meantime, legitimate users log

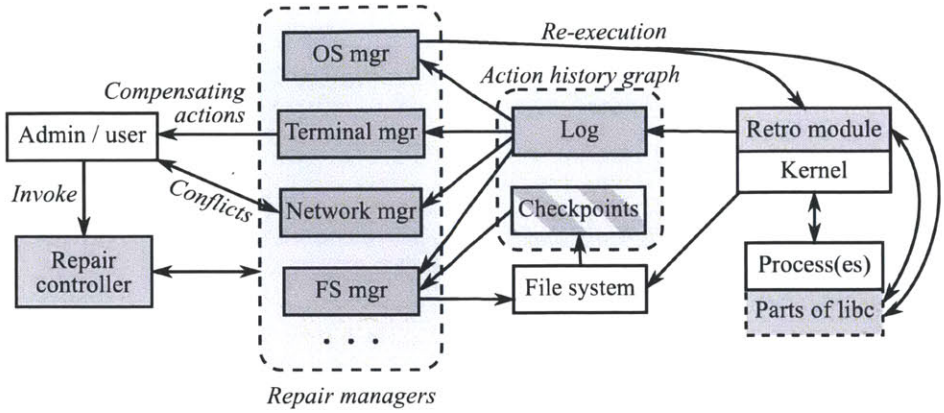

Figure 2-1: Overview of RETRO's architecture, including major components and their interactions. Shading indicates components introduced by RETRO. Striped shading of checkpoints indicates that RETRO reuses existing file system snapshots when available.

in, invoke their own PHP scripts, use `texi2pdf`, and root adds new legitimate users.

To undo attacks, RETRO provides a system-wide architecture for recording actions, causes, and effects in order to identify all the downstream effects of a compromise. The key challenge is that a compromise in the past may have effects on subsequent legitimate actions, especially if the administrator discovers an attack long after it occurred. RETRO must sort out this entanglement automatically and efficiently. In our running example, Eve's changes to the password file and to `texi2pdf` are entangled with legitimate actions that modified or accessed the password file, or used `texi2pdf`. If legitimate users ran `texi2pdf`, their output depended on Eve's actions, and so did any programs that used that output in turn.

One main difference from most previous systems (more details in §8) is that, unlike RETRO, they require user input to disentangle such actions. Typical previous solutions are good at detecting a compromise and allow a user to roll the system back to a check point before the compromise, but then ask the user to incorporate legitimate changes from after the compromise manually; this can be quite onerous if the attack has happened a long time ago. Some solutions reduce the amount of manual work for special cases (e.g., known viruses). The most recent general solution for reducing user assistance (Taser [20]) incurs many false positives (undoing legitimate actions), or, after white-listing some actions to minimize false positives, it incurs false negatives (missing parts of the attack).

2.2 Normal execution

As the computer executes, RETRO must record sufficient information to be able to revert the effects of an attack. To this end, RETRO records periodic checkpoints of persistent state (the file system), so that it can later roll back to a checkpoint. RETRO does not require any specialized format for its file system checkpoints; if the file system already creates periodic snapshots, such as [31, 37, 43, 44], RETRO can simply use these snapshots, and requires no checkpointing of its own. In addition to rollback, RETRO must be able to re-execute affected computations. To this end, RETRO logs actions executed over time, along with their dependencies. The resulting checkpoints and actions comprise RETRO's *action history graph*, such as the one shown in Figure 3-1.

The action history graph consists of two kinds of objects: *data objects*, such as files, and *actor objects*, such as processes. Each object has a set of checkpoints, representing a copy of its state at different points in time. Each actor object additionally consists of a set of *actions*, representing the execution of that actor over some period of time. Each action has dependencies from and to other objects in the graph, representing the objects accessed and modified by that action. Actions and checkpoints of adjacent objects are ordered with respect to each other, in the order in which they occurred.[1]

RETRO stores the action history graph in a series of log files over time. When RETRO needs more space for new log files, it garbage-collects older log files (by deleting them). Log files are only useful to RETRO in conjunction with a checkpoint that precedes the log files, so log files with no preceding checkpoint can be garbage-collected. In practice, this means that RETRO keeps checkpoints for at least as long as the log files. By design, RETRO cannot recover from an intrusion whose log files have been garbage collected; thus, the amount of log space allocated to logs and checkpoints controls RETRO's recovery "horizon". For example, a web server running the HotCRP paper review software [28] logs 4 GB of data per day, so if the administrator dedicates a 2 TB disk ($100) to RETRO, he or she can recover from attacks within the past year, although these numbers strongly depend on the application.

[1] For simplicity, our prototype globally orders all checkpoints and actions for all objects.

2.3 Intrusion detection

At some point after an adversary compromises the system, the system administrator learns of the intrusion, perhaps with the help of an intrusion detection system. To repair from the intrusion, the system administrator must first track down the initial intrusion point, such as the adversary's network connection, or a user accidentally running a malware binary. RETRO provides a tool similar to BackTracker [26] that helps the administrator find the intrusion point, starting from the observed symptoms, by leveraging RETRO's action history graph. In the rest of this thesis, we assume that an intrusion detection system exists, and we do not describe our BackTracker-like tool in any more detail.

2.4 Repair

Once the administrator finds the intrusion point, he or she reboots the system, to discard non-persistent state, and invokes RETRO's repair controller, specifying the name of the intrusion point determined in the previous step.[2] The repair controller undoes the offending action, A, by rolling back objects modified by A to a previous checkpoint, and replacing A with a no-op in the action history graph. Then, using the action history graph, the controller determines which other actions were potentially influenced by A (e.g., the values of their arguments changed), rolls back the objects they depend on (e.g., their arguments) to a previous checkpoint, re-executes those actions in their corrected environment (e.g., with the rolled-back arguments), and then repeats the process for actions that the re-executed actions may have influenced. This process will also undo subsequent actions by the adversary, since the action that initially caused them, A, has been reverted. Thus, after repair, the system will contain the effects of all legitimate actions since the compromise, but none of the effects of the attack.

To minimize re-execution and to avoid potential conflicts, the repair controller checks whether the inputs to each action are semantically equivalent to the inputs during original

[2]Each object and action in the action history graph has a unique name, as described in §4.

execution, and skips re-execution in that case. In our running example, if Alice's sshd process reads a password file that Eve modified, it might not be necessary to re-execute sshd if its execution only depended on Alice's password entry, and Eve did not change that entry. If Alice's sshd later changed her password entry, then this change will not result in a conflict during repair because the repair controller will determine that her change to the password file could not have been influenced by Eve.

RETRO's repair controller must manipulate many kinds of objects (e.g., files, directories, processes, etc.) and re-execute many types of actions (e.g., system calls and function calls) during repair. To ensure that RETRO's design is extensible, RETRO's action history graph provides a well-defined API between the repair controller and individual graph objects and actions. Using this API, the repair controller implements a generic repair algorithm, and interacts with the graph through individual *repair managers* associated with each object and action in the action history graph. Each repair manager, in turn, tracks the state associated with their respective object or action, implements object/action-specific operations during repair, and efficiently stores and accesses the on-disk state, logs, and checkpoints.

2.5 External dependencies

During repair, RETRO may discover that changes made by the adversary were externally visible. RETRO relies on compensating actions to deal with external dependencies where possible. For example, if a user's terminal output changes, RETRO sends a diff between the old and new terminal sessions to the user in question.

In some cases, RETRO does not have a compensating action to apply. If Eve, from our running example, connected to her botnet client over the network, RETRO would not be able to re-execute the connection during repair (the connection will be refused since the botnet will no longer be running). When such a situation arises, RETRO's repair controller pauses re-execution and asks the administrator to manually re-execute the appropriate action. In the case of Eve's connection, the administrator can safely do nothing and tell the repair controller to resume.

2.6 Assumptions

RETRO makes three significant assumptions. First, RETRO assumes that the system administrator detects intrusions in a timely manner, that is, before the relevant logs are garbage-collected. An adversary that is aware of RETRO could compromise the system and then try to avoid detection, by minimizing any activity until RETRO garbage-collects the logs from the initial intrusion. If the initial intrusion is not detected in time, the administrator will not be able to revert it directly, but this strategy would greatly slow down attackers. Moreover, the administrator may be able to revert subsequent actions by the adversary that leveraged the initial intrusion to cause subsequent notable activity.

Second, RETRO assumes that the administrator promptly detects any intrusions with wide-ranging effects on the execution of the entire system. If such intrusions persist for a long time, RETRO will require re-execution of large parts of the system, potentially incurring many conflicts and requiring significant user input. However, we believe this assumption is often reasonable, since the goal of many adversaries is to remain undetected for as long as possible (e.g., to send more spam, or to build up a large botnet), and making pervasive changes to the system increases the risk of detection.

Third, we assume that the adversary compromises a computer system through user-level services. The adversary may install new programs, add backdoors to existing programs, modify persistent state and configuration files, and so on, but we assume the adversary doesn't tamper with the kernel, file system, checkpoints, or logs. RETRO's techniques rely on a detailed understanding of operating system objects, and our assumptions allow RETRO to trust the kernel state of these objects. We rely on existing techniques for hardening the kernel, such as [19, 33, 45, 47], to achieve this goal in practice.

Chapter 3

Action history graph

RETRO's design centers around the *action history graph*, which represents the execution of the entire system over time. The action history graph must address four requirements in order to disentangle attacker actions from legitimate operations. First, it must operate *system-wide*, capturing all dependencies and actions, to ensure that RETRO can detect and repair all effects of an intrusion. Second, the graph must support *fine-grained re-execution* of just the actions affected by the intrusion, without having to re-execute unaffected actions. Third, the graph must be able to *disambiguate attack actions* from legitimate operations whenever possible, without introducing false dependencies. Finally, recording and accessing the action history graph must be *efficient*, to reduce both runtime overheads and repair time. The rest of this section describes the design of RETRO's action history graph.

3.1 Repair using the action history graph

RETRO represents an attack as a set of *attack actions*. For example, an attack action can be a process reading data from the attacker's TCP connection, a user inadvertently running malware, or an offending file write. Given a set of attack actions, RETRO repairs the system in two steps, as follows.

First, RETRO replaces the attack actions with benign actions in the action history graph. For example, if the attack action was a process reading a malicious request from

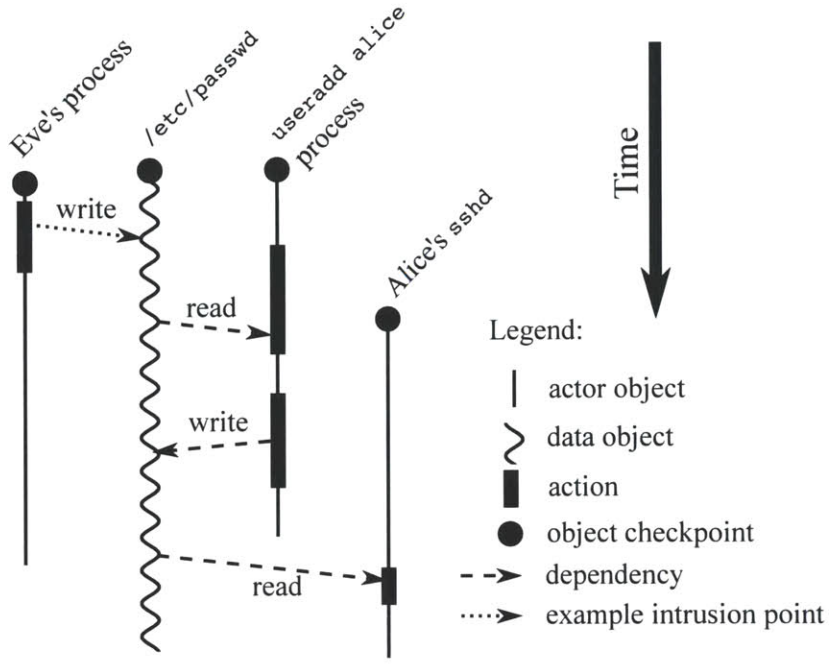

Figure 3-1: A simplified view of the action history graph depicting Eve's attack in our running example. In this graph, attacker Eve adds an account for herself to `/etc/passwd`, after which root adds an account for Alice, and Alice logs in via ssh. As an example, we consider Eve's write to the password file to be the attack action, although in reality, the attack action would likely be the network connection that spawned Eve's process in the first place. Not shown are intermediate data objects, and system call actors, described in §3.3 and Figure 3-2.

the attacker's TCP connection, RETRO removes the request data, as if the attacker never sent any data on that connection. If the attack action was a user accidentally running malware, RETRO changes the user's `exec` system call to run `/bin/true` instead of the malware binary. Finally, if the attack action was an unwanted write to a file, as in Figure 3-1, RETRO replaces the action with a zero-byte write. RETRO includes a handful of such benign actions used to neutralize intrusion points found by the administrator.

Second, RETRO repairs the system state to reflect the above changes, by iteratively re-executing affected actions, starting with the benign replacements of the attack actions themselves. Prior to re-executing an action, RETRO must roll back all input and output objects of that action, as well as the actor itself, to an earlier checkpoint. For example, in Figure 3-1, RETRO rolls back the output of the attack action—namely, the password file object—to its earlier checkpoint.

RETRO then considers all actions with dependencies to or from the objects in question, according to their time order. Actions with dependencies *to* the object in question are

Function or variable		Semantics
set⟨checkpt⟩	object→checkpts	Set of available checkpoints for this object.
void	object→rollback(*c*)	Roll back this object to checkpoint *c*.
set⟨action⟩	actor_obj→actions	Set of actions that comprise this actor object.
set⟨action⟩	data_obj→readers	Set of actions that have a dependency from this data object.
set⟨action⟩	data_obj→writers	Set of actions that have a dependency to this data object.
set⟨data_obj⟩	data_obj→parts	Set of data objects whose state is part of this data object.
actor_obj	action→actor	Actor containing this action.
set⟨data_obj⟩	action→inputs	Set of data objects that this action depends on.
set⟨data_obj⟩	action→outputs	Set of data objects that depend on this action.
bool	action→equiv()	Check whether any inputs of this action have changed.
bool	action→connect()	Add dependencies for new inputs and outputs, based on new inputs.
void	action→redo()	Re-execute this action, updating output objects.

Table 3.1: Object (top) and action (bottom) repair manager API.

re-executed, to reconstruct the object. For actions with dependencies *from* the object in question, RETRO checks whether their inputs are semantically equivalent to their inputs during original execution. If the inputs are different, such as the `useradd` command reading the modified password file in Figure 3-1, the action will be re-executed, following the same process as above. On the other hand, if the inputs are semantically equivalent, RETRO skips re-execution, avoiding the repair cascade. For example, re-executing `sshd` may be unnecessary, if the password file entry accessed by `sshd` is the same before and after repair. We will describe shortly how RETRO determines this (in §3.4 and Figure 3-3).

3.2 Graph API

As described above, repairing the system requires three functions: rolling back objects to a checkpoint, re-executing actions, and checking an action's input dependencies for semantic equivalence. To support different types of objects and actions in a system-wide action history graph, RETRO delegates these tasks, as well as tracking the graph structure itself, to *repair managers* associated with each object and action in the graph.

A manager consists of two halves: a runtime half, responsible for recording logs and checkpoints during normal execution, and a repair-time half, responsible for repairing the system state once the system administrator invokes RETRO to repair an intrusion.

The runtime half has no pre-defined API, and needs to only synchronize its log and checkpoint format with the repair-time half. On the other hand, the repair-time half has a well-defined API, shown in Table 3.1.

Object manager. During normal execution, object managers are responsible for making periodic checkpoints of objects. For example, the file system manager takes snapshots of files, such as a copy of `/etc/passwd` in Figure 3-1. Process objects also have checkpoints in the graph, although in our prototype, the only supported process checkpoint is the initial state of a process immediately prior to `exec`.

During repair, an object manager is responsible for maintaining the state represented by its object. For persistent objects, the manager uses the on-disk state, such as the actual file for a file object. For ephemeral objects, such as processes or pipes, the manager keeps a temporary in-memory representation to help action managers redo actions and check predicates, as we describe in §4.

An object manager provides one main procedure invoked during repair, $o \rightarrow \mathit{rollback}(v)$, which rolls back object o's state to checkpoint v. For a file object, this means restoring the on-disk file from snapshot v. For a process, this means constructing an initial, paused process in preparation for redoing `exec`, as we will discuss in §4.2; since there is only one kind of process checkpoint, v is not used. If the object was last checkpointed long ago, RETRO will need to re-execute all subsequent actions that modified the data object, or that comprise the actor object.

Action manager. During normal execution, action managers are responsible for recording all actions executed by actors in the system. For each action, the manager records enough information to re-execute the same action at repair time, as well as to check whether the inputs are semantically equivalent (e.g., by recording the data read from a file).

At repair time, an action manager provides three procedures. First, $a \rightarrow \mathit{redo}()$ re-executes action a, reading new data from a's input objects and modifying the state of a's output objects. For example, redoing a file write action modifies the corresponding

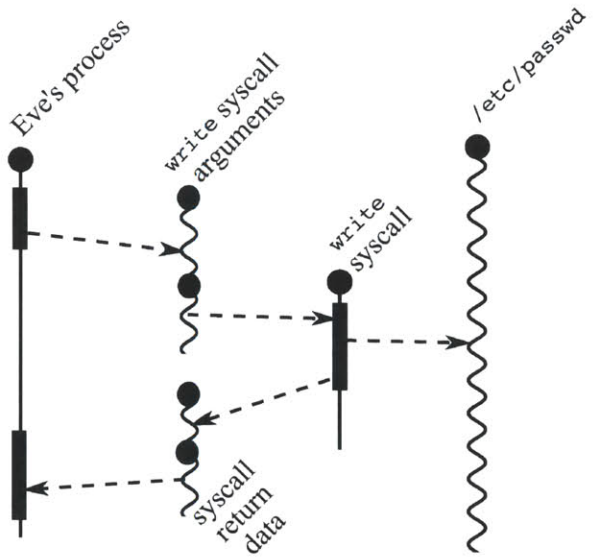

Figure 3-2: An illustration of the system call actor object and arguments and return value data objects, for Eve's write to the password file from Figure 3-1. Legend is the same as in Figure 3-1.

file in the file system; if the action was not otherwise modified, this would write the same data to the same offset as during original execution. Second, $a \rightarrow equiv()$ checks whether a's inputs have semantically changed since the original execution. For instance, *equiv* on a file read action checks whether the file contains the same data at the same offset (and, therefore, whether the read call would return the same data). Finally, $a \rightarrow connect()$ updates action a's input and output dependencies, in case that changed inputs result in the action reading or modifying new objects. To ensure that past dependencies are not lost, *connect* only adds, and never removes, dependencies (even if the action in question does not use that dependency).

3.3 Refining actor objects: finer-grained re-execution

An important goal of RETRO's design is minimizing re-execution, so as to avoid the need for user input to handle potential conflicts and external dependencies. It is often necessary to re-execute a subset of an actor's actions, but not necessarily the entire actor. For example, after rolling back a file like `/etc/passwd` to a checkpoint that was taken long ago, RETRO needs to replay all writes to that file, but should not need to re-execute the processes that issued those writes. Similarly, in Figure 3-1, RETRO would ideally re-

execute only a part of `sshd` that checks whether Alice's password entry is the same, and if so, avoid re-executing the rest of `sshd`, which would lead to an external dependency because cryptographic keys would need to be re-negotiated. Unfortunately, re-executing a process from an intermediate state is difficult without process checkpointing.

To address this challenge, RETRO *refines* actors in the action history graph to explicitly denote parts of a process that can be independently re-executed. For example, RETRO models every system call issued by a process by a separate system call actor, comprising a single system call action, as shown in Figure 3-2. The system call arguments, and the result of the system call, are explicitly represented by system call argument and return value objects. This allows RETRO to re-execute individual system calls when necessary (e.g., to re-construct a file during repair), while avoiding re-execution of entire processes if the return values of system calls remain the same.

The same technique is also applied to re-execute specific functions instead of an entire process. Figure 3-3 shows a part of the action history graph for our running example, in which `sshd` creates a separate actor to represent its call to `getpwnam("alice")`. While `getpwnam`'s execution depends on the entire password file, and thus must be re-executed if the password file changes, its return value contains only Alice's password entry. If re-execution of `getpwnam` produces the same result, the rest of `sshd` need not be re-executed. §4 describes such higher-level managers in more detail.

The same mechanism helps RETRO create benign replacements for attack actions. For example, in order to undo a user accidentally executing malware, RETRO changes the `exec` system call's arguments to invoke `/bin/true` instead of the malware binary. To do this, RETRO synthesizes a new checkpoint for the object representing `exec`'s arguments, replacing the original malware binary path with `/bin/true`, and rolls back that object to the newly-created "checkpoint", as illustrated in Figure 3-5 and §3.5.

3.4 Refining data objects: finer-grained dependencies

While OS-level dependencies ensure completeness, they can be too coarse-grained, leading to false dependencies, such as every process depending on the `/tmp` directory.

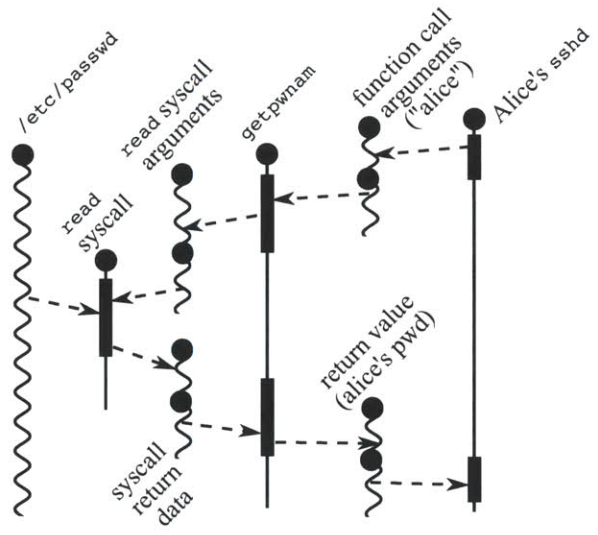

Figure 3-3: An illustration of refinement in an action history graph, depicting the use of additional actors to represent a re-executable call to `getpwnam` from `sshd`. Legend is the same as in Figure 3-1.

RETRO's design addresses this problem by *refining* the same state at different levels of abstraction in the graph when necessary. For instance, a directory manager creates individual objects for each file name in a directory, and helps disambiguate directory lookups and modifications by recording dependencies on specific file names.

The challenge in supporting refinement in the action history graph lies in dealing with multiple objects representing the same state. For example, the state of a single directory entry is a part of both the directory manager's object for that specific file name, as well as the file manager's node for that directory's inode. On one hand, we would like to avoid creating dependencies to and from the underlying directory inode, to prevent false dependencies. On the other hand, if some process does directly read the underlying directory inode's contents, it should depend on all of the directory entries in that directory.

To address this challenge, each object in RETRO keeps track of other objects that represent parts of its state. For example, the manager of each directory inode keeps track of all the directory entry objects for that directory. The object manager exposes this set of parts through the $o \rightarrow parts$ property, as shown in Table 3.1. In most cases, the manager tracks its parts through hierarchical names, as we discuss in §4.

RETRO's OS manager records all dependencies, even if the same dependency is also recorded by a higher-level manager. This means that RETRO can determine trust

```
function ROLLBACK(node, checkpt)
    node → rollback(checkpt)
    state[node] := checkpt
end function

function PREPAREREDO(action)
    if ¬action → connect() then return FALSE
    end if
    if state[action → actor] > action then
        cps := action → actor → checkpts
        cp := max(c ∈ cps | c ≤ action)
        ROLLBACK(action → actor, cp)
        return FALSE
    end if
    for all o ∈ (action → inputs ∪ action → outputs) do
        if state[o] ≤ action then continue
        end if
        ROLLBACK(o, max(c ∈ o → checkpts | c ≤ action))
        return FALSE
    end for
    return TRUE
end function

function PICKACTION()
    actions := ∅
    for all o ∈ state | o is actor object do
        actions += min(a ∈ o → actions | a > state[o])
    end for
    for all o ∈ state | o is data object do
        actions += min(a ∈ o → readers ∪ o → writers | a > state[o])
    end for
    return min(actions)
end function
```

Figure 3-4: The repair algorithm (a).

in higher-level dependencies at repair time. If the appropriate manager mediated all modifications to the larger object (such as a directory inode), and the manager was not compromised, RETRO can safely use finer-grained objects (such as individual directory entry objects). Otherwise, RETRO uses coarse-grained but safe OS-level dependencies.

```
function RepairLoop()
    while a := PickAction() do
        if a → equiv() and state[o] ≥ a,
            ∀o ∈ a → outputs ∪ a → actor then
            for all i ∈ a → inputs ∩ keys(state) do
                state[i] := a
            end for
            continue                          ▷ skip semantically-equivalent action
        end if
        if PrepareRedo(a) then
            a → redo()
            for all o ∈ a → inputs ∪ a → outputs ∪ a → actor do
                state[o] := a
            end for
        end if
    end while
end function

function Repair(repair_obj, repair_cp)
    Rollback(repair_obj, repair_cp)
    RepairLoop( )
end function
```

Figure 3-5: The repair algorithm (b).

3.5 Repair controller

Retro uses a *repair controller* to repair system state with the help of object and action managers. Figure 3-5 summarizes the pseudo-code for the repair controller. The controller, starting from the Repair function, creates a parallel "repaired" timeline by re-executing actions in the order that they were originally executed. To do so, the controller maintains a set of objects that it is currently repairing (the *nodes* hash table), along with the last action that it performed on that object. RepairLoop continuously attempts to re-execute the next action, until it has considered all actions, at which point the system state is fully repaired.

To choose the next action for re-execution, RepairLoop invokes PickAction, which chooses the earliest action that hasn't been re-executed yet, out of all the objects being repaired. If the action's inputs are the same (according to *equiv*), and none of the outputs of the action need to be reconstructed, RepairLoop does not re-execute the action, and

just advances the state of the action's input nodes. If the action needs to be re-executed, RepairLoop invokes PrepareRedo, which ensures that the action's actor, input objects, and output objects are all in the right state to re-execute the action (by rolling back these objects when appropriate). Once PrepareRedo indicates it is ready, RepairLoop re-executes the action and updates the state of the actor, input, and output objects. Finally, Repair invokes RepairLoop in the first place, after rolling back *repair_obj* to the (newly-synthesized) checkpoint *repair_cp*, as described in §3.3.

Not shown in the pseudo-code is handling of refined objects. When the controller rolls back an object that has a non-empty set of parts, it must consider re-executing actions associated with those parts, in addition to actions associated with the larger object. Also not shown is the checking of integrity for higher-level dependencies, as described in §3.4.

Chapter 4

Object and action managers

This section describes RETRO's object and action managers, starting with the file system and OS managers that guarantee completeness of the graph, and followed by higher-level managers that provide finer-grained dependencies for application-specific parts of the graph.

4.1 File system manager

The file system manager is responsible for all file objects. To uniquely identify files, the manager names file objects by $\langle device, part, inode \rangle$. The *device* and *part* components identify the disk and partition holding the file system. Our current prototype disallows direct access to partition block devices, so that file system dependencies are always trusted. The *inode* number identifies a specific file by inode, without regard to path name. To ensure that files can be uniquely identified by inode number, the file system manager prevents inode reuse until all checkpoints and logs referring to the inode have been garbage-collected.

During normal operation, the file system manager must periodically checkpoint its objects (including files and directories), using any checkpointing strategy. Our implementation relies on a snapshotting file system to make periodic snapshots of the entire file system tree (e.g., once per day). This works well for systems which already create daily snapshots [31, 37, 43, 44], where the file system manager can simply

leverage existing snapshots. Upon file deletion, the file system manager moves the deleted inode into a special directory, so that it can reuse the same exact inode number on rollback. The manager preserves the inode's data contents, so that RETRO can undo an unlink operation by simply linking the inode back into a directory (see §4.3).

During repair, the file system manager's *rollback* method uses a special kernel module to open the checkpointed file as well as the current file by their inode number. Once the repair manager obtain a file descriptor for both inodes, it overwrites the current file's contents with the checkpoint's contents, or re-constructs an identical set of directory entries, for directory inodes. On rollback to a file system snapshot where the inode in question was not allocated yet, the file system manager truncates the file to zero bytes, as if it was freshly created. As a precaution, the file system manager creates a new file system snapshot before initiating any rollback.

4.2 OS manager

The OS manager is responsible for process and system call actors, and their actions. The manager names each process in the graph by $\langle bootgen, pid, pidgen, execgen \rangle$. *bootgen* is a boot-up generation number to distinguish process IDs across reboots. *pid* is the Unix process ID, and *pidgen* is a generation number for the process ID, used to distinguish recycled process IDs. Finally, *execgen* counts the number of times a process called the `exec` system call; the OS manager logically treats `exec` as creating a new process, albeit with the same process ID. The manager names system calls by $\langle bootgen, pid, pidgen, execgen, sysid \rangle$, where *sysid* is a per-process unique ID for that system call invocation.

Recording normal execution

During normal execution, the OS manager intercepts and records all system calls that create dependencies to or from other objects (i.e., not `getpid`, etc), recording enough information about the system calls to both re-execute them at repair time, and to check whether the inputs to the system call are semantically equivalent. The OS manager creates nominal checkpoints of process and system call actors. Since checkpointing of

processes mid-execution is difficult [15, 41], our OS manager checkpoints actors only in their "initial" state immediately prior to `exec`, denoted by $\perp$. The OS manager also keeps track of objects representing ephemeral state, including pipes and special devices such as `/dev/null`. Although RETRO does not attempt to repair this state, having these objects in the graph helps track and check dependencies using *equiv* during repair, and to perform partial re-execution.

Action history graph representation

In the action history graph, the OS manager represents each system call by two actions in the process actor, two intermediate data objects, and a system call actor and action, as shown in Figure 3-2. The first process action, called the *syscall invocation* action, represents the execution of the process up until it invokes the system call. This action conceptually places the system call arguments, and any other relevant state, into the system call arguments object. For example, the arguments for a file write include the target inode, the offset, and the data. The arguments for `exec`, on the other hand, include additional information that allows re-executing the system call actor without having to re-execute the process actor, such as the current working directory, file descriptors not marked `O_CLOEXEC`, and so on.

The system call action, in a separate actor, conceptually reads the arguments from this object, performs the system call (incurring dependencies to corresponding objects), and writes the return value and any returned data into the return value object. For example, a write system call action, shown in Figure 3-2, creates a dependency to the modified file, and stores the number of bytes written into the return value object. Finally, the second process action, called the *syscall return* action, reads the returned data from that object, and resumes process execution. In case of `fork` or `exec`, the OS manager creates two return objects and two syscall return actions, representing return values to both the old and new process actors. Thus, every process actor starts with a syscall return action, with a dependency from the return object for `fork` or `exec`.

In addition to system calls, Unix processes interact with memory-mapped files. RETRO cannot re-execute memory-mapped file accesses without re-executing the process. Thus,

the OS manager associates dependencies to and from memory-mapped files with the process's own actions, as opposed to actions in a system call actor. In particular, every process action (either syscall invocation or return) has a dependency *from* every file memory-mapped by the process at that time, and a dependency *to* every file memory-mapped as writable at that time.

Shepherded re-execution

During repair, the OS manager must re-execute two types of actors: process actors and system call actors. For system call actors, when the repair controller invokes *redo*, the OS manager reads the (possibly changed) values from the system call arguments object, executes the system call in question, and places return data into the return object. *equiv* on a system call action checks whether the input objects have the same values as during the original execution. Finally, *connect* reads the (possibly changed) inputs, and creates any new dependencies that result. For example, if a `stat` system call could not find the named file during original execution, but RETRO restores the file during repair, *connect* would create a new dependency from the newly-restored file.

For process actors, the OS manager represents the state of a process during repair with an actual process being *shepherded* via the `ptrace` debug interface. On $p \rightarrow \mathit{rollback}(\perp)$, the OS manager creates a fresh process for process object p under `ptrace`. When the repair controller invokes *redo* on a syscall return action, the OS manager reads the return data from the corresponding system call return object, updates the process state using `PTRACE_POKEDATA` and `PTRACE_SETREGS`, and allows the process to execute until it's about to invoke the next system call. *equiv* on a system call return action checks if the data in the system call return object is the same as during the original execution. When the repair controller invokes *redo* on the subsequent syscall invocation action, the OS manager simply marshals the arguments for the system call invocation into the corresponding system call arguments object. This allows the repair controller to separately schedule the re-execution of the system call, or to re-use previously recorded return data. Finally, *connect* does nothing for process actions.

One challenge for the OS manager is to deal with processes that issue different system

calls during re-execution. The challenge lies in matching up system calls recorded during original execution with system calls actually issued by the process during re-execution. The OS manager employs greedy heuristics to match up the two system call streams. If a new syscall does not match a previously-recorded syscall in order, the OS manager creates new system call actions, actors, and objects (as shown in Figure 3-2). Similarly, if a previously-recorded syscall does not match the re-executed system calls in order, the OS manager replaces the previously-recorded syscall's actions with no-ops. In the worst case, the only matches will be the initial return from `fork` or `exec`, and the final syscall invocation that terminates the process, potentially leading to more re-execution, but not a loss of correctness.

In our running example, Eve trojans the `texi2pdf` shell script by adding an extra line to start her botnet worker. After repairing the `texi2pdf` file, RETRO re-executes every process that ran the trojaned `texi2pdf`. During shepherded re-execution of `texi2pdf`, `exec` system calls to legitimate LaTeX programs are identical to those during the original execution; in other words, the system call argument objects are equivalent, and *equiv* on the system call action returns true. As a result, there is no need to re-execute these child processes. However, `exec` system calls to Eve's bot are missing, so the manager replaces them with no-ops, which recursively undoes any changes made by Eve's bot.

4.3 Directory manager

The directory manager is responsible for exposing finer-grained dependency information about directory entries. Although the file system manager tracks changes to directories, it treats the entire directory as one inode, causing false dependencies in shared directories like `/tmp`. The directory manager names each directory entry by $\langle device, part, inode, name \rangle$. The first three components of the name are the file system manager's name for the directory inode. The *name* part represents the file name of the directory entry.

During normal operation, the directory manager must record checkpoints of its objects, conceptually consisting of the inode number for the directory entry (or $\perp$ to

represent non-existent directory entries). However, since the file system manager already records checkpoints of all directories, the directory manager relies on the file system manager's checkpoints, and does not perform any checkpointing of its own. The directory manager similarly relies on the OS manager to record dependencies between system call actions and directory entries accessed by those system calls, such as name lookups in `namei` (which incur a dependency from every directory entry traversed), or directory modifications by `rename` (which incur a dependency to the modified directory entries).

During repair, the directory manager's sole responsibility is rolling back directory entries to a checkpoint; the OS manager handles redo of all system calls. To roll back a directory entry to an earlier checkpoint, the directory manager finds the inode number contained in that directory entry (using the file system manager's checkpoint), and changes the directory entry in question to point to that inode, with the help of RETRO's kernel module. If the directory entry did not exist in the checkpoint, the directory manager similarly unlinks the directory entry.

4.4 System library managers

Every user login on a typical Unix system accesses several system-wide files. For example, each login attempt accesses the entire password file, and successful logins update both the `utmp` file (tracking currently logged in users) and the `lastlog` file (tracking each user's last login). In a naïve system, these shared files can lead to false dependencies, making it difficult to disambiguate attacker actions from legitimate changes. To address this problem, RETRO uses a *libc* system library manager to expose the semantic independence between these actions.

One strawman approach would be to represent such shared files much as directories (i.e., creating a separate object for each user's password file entry). However, unlike the directory manager, which mediates all accesses to a directory, a manager for a function in *libc* cannot guarantee that an attacker will not bypass it—the manager, libc, and the attacker can be in the same address space. Thus, the `libc` manager does not change the representation of data objects, and instead simplifies re-execution, by creating actors to

represent the execution of individual *libc* functions. For example, Figure 3-3 shows an actor for the `getpwnam` function call as part of `sshd`.

During normal operation, the library manager creates a fresh actor for each function call to one of the managed functions, such as `getpwnam`, `getspnam`, and `getgrouplist`. The library manager names function call actors by $\langle bootgen, pid, pidgen, execgen, callgen \rangle$; the first four parts name the process, and *callgen* is a unique ID for each function call. Much as with system call actors, the arguments object contains the function name and arguments, and the return object contains the return value. Like processes, function call actors have only one checkpoint, $\perp$, representing their initial state prior to the call.

The library manager requires the OS manager's help to associate system calls issued from inside library functions with the function call actor, instead of the process actor. To do this, the OS manager maintains a "call stack" of function call actors that are currently executing. On every function call, the library manager pushes the new function call actor onto the call stack, and on return, it pops the call stack. The OS manager associates syscall invocation and return actions with the last actor on the call stack, if any, instead of the process actor.

During repair, the library manager's *rollback* and *redo* methods allow the repair controller to re-execute individual functions. For example, in Figure 3-3, the controller will re-execute `getpwnam`, because its dependency on `/etc/passwd` changed due to repair. However, if *equiv* indicates the return value from `getpwnam` did not change, the controller need not re-execute the rest of `sshd`.

RETRO's trust assumption about the library manager is that the function does not semantically affect the rest of the program's execution other than through its return value. If an attacker process compromises its own *libc* manager, this does not pose a problem, because the process already depended on the attacker in other ways, and RETRO will repair it. However, if an attacker exploits a vulnerability in the function's input parsing code (such as a buffer overflow in `getpwnam` parsing `/etc/passwd`), it can take control of `getpwnam`, and influence the execution of the process in ways other than `getpwnam`'s return value. Thus, RETRO trusts *libc* functions wrapped by the library manager to safely parse files and faithfully represent their return values.

4.5 Terminal manager

Undoing attacker's actions during repair can result in legitimate applications sending different output to a user's terminal. For example, if the user ran `ls /tmp`, the output may have included temporary files created by the attacker, or the `ls` binary was trojaned by the attacker to hide certain files. While RETRO cannot undo what the user already saw, the terminal manager helps RETRO generate compensating actions.

The terminal manager is responsible for objects representing pseudo-terminal, or pty, devices (`/dev/pts/`*N* in Linux). During normal operation, the manager records the user associated with each pty (with help from `sshd`), and all output sent to the pty. During repair, if the output sent to the pty differs from the output recorded during normal operation, the terminal manager computes a text diff between the two outputs, and emails it to the user.

4.6 Network manager

The network manager is responsible for compensating for externally-visible changes. To this end, the network manager maintains objects representing the outside world (one object for each TCP connection, and one object for each IP address/UDP port pair). During normal operation, the network manager records all traffic, similar to the terminal manager.

During repair, the network manager compares repaired outgoing data with the original execution. When the network manager detects a change in outgoing traffic, it flags an external dependency, and presents the user or administrator with three choices. The first choice is to ignore the dependency, which is appropriate for network connections associated with the adversary (such as Eve's login session in our running example, which will generate different network traffic during repair). The second choice is to re-send the network traffic, and wait for a response from the outside world. This is appropriate for outgoing network connections and idempotent protocols, such as DNS. Finally, the third choice is to require the user to manually resolve the external dependency, such as

by manually re-playing the traffic for incoming connections. This is necessary if, say, the response to an incoming SMTP connection has changed, the application did not provide its own compensating action, and the user does not want to ignore this dependency.

Chapter 5

Implementation

We implemented a prototype of RETRO for Linux,[1] components of which are summarized in Table 5.1. During normal execution, a kernel module intercepts and records all system calls to a log file, implementing the runtime half of the OS, file system, directory, terminal, and network managers. To allow incremental loading of log records, RETRO records an index alongside the log file that allows efficient lookup of records for a given process ID or inode number. The file system manager implements checkpoints using subvolume snapshots in btrfs [43]. The libc manager logs function calls using a new RETRO system call to add ordered records to the system-wide log. The repair controller, and the repair-time half of each manager, are implemented as Python modules.

RETRO implements three optimizations to reduce logging costs. First, it records SHA-1 hashes of data read from files, instead of the actual data. This allows checking for equivalence at repair time, but avoids storing the data twice. Second, it does not record data read or written by white-listed deterministic processes (in our prototype, this includes `gcc` and `ld`). This means that, if any of the read or write dependencies to or from these processes are suspected during repair, the entire process will have to be re-executed, because individual read and write system calls cannot be checked for equivalence or re-executed. Since all of the dependency relationships are preserved, this optimization trades off repair time for recording time, but does not compromise

[1] While our prototype is Linux-specific, we believe that RETRO's approach is equally applicable to other operating systems.

Component	Lines of code
Logging kernel module	3,300 lines of C
Repair controller, manager modules	5,000 lines of Python
System library managers	700 lines of C
Backtracking GUI tool	500 lines of Python

Table 5.1: Components of our RETRO prototype, and an estimate of their complexity, in terms of lines of code.

completeness. Third, RETRO compresses the resulting log files to save space.

Chapter 6

Evaluation

This section answers three questions about RETRO, in turn. First, what kinds of attacks can RETRO recover from, and how much user input does it require? Second, are all of RETRO's mechanisms necessary in practice? And finally, what are the performance costs of RETRO, both during normal execution and during repair?

6.1 Recovery from attack

To evaluate how RETRO recovers from different attacks, we used three classes of attack scenarios. First, to make sure we can repair real-world attacks, we used attacks recorded by a honeypot. Second, to make sure RETRO can repair worst-case attacks, we used synthetic attacks designed to be particularly challenging for RETRO, including the attack from our running example. For both real-world and synthetic attacks, we perform user activity described in the running example after the attack takes place—namely, root logs in via ssh and adds an account for Alice, who then also logs in via ssh to edit and build a LaTeX file. Finally, we compare RETRO to Taser, the state-of-the-art attack recovery system, using attack scenarios from the Taser paper [20].

Honeypot attacks. To collect real-world attacks, we ran a honeypot [1] for three weeks, with a modified `sshd` that accepted any password for login as root. Out of many root logins, we chose two attacks that corrupted our honeypot's state in the most

Attack	Objects repaired with predicates			Objects repaired without predicates			User input
	Proc	Func	File	Proc	Func	File	
Password change	1	2	4	430	20	274	1
Log cleaning	59	0	40	60	0	40	0
Running example	58	57	75	513	61	300	1
sshd trojan	530	47	303	530	47	303	3

Table 6.1: Repair statistics for the two honeypot attacks (top) and two synthetic attacks (bottom). The repaired objects are broken down into processes, functions (from libc), and files. Intermediate objects such as syscall arguments are not shown. The concurrent workload consisted of 1,261 process, function, and file objects (both actor and data objects), and 16,239 system call actions. RETRO was able to fully repair all attacks, with no false positives or false negatives. User input indicate the number of times RETRO asked for user assistance in repair; the nature of the conflict is reported in §6.

Scenario	Taser				RETRO
	Snapshot	NoI	NoIAN	NoIANC	
Illegal storage	FP	FP	FN	FN	✓
Content destruction	FP	✓	✓	FN	✓
Unhappy student	FP	FP	✓	FN	✓
Compromised database	FP	FP	FP	FN	✓
Software installation	FP	FP	✓	✓	✓
Inexperienced admin	FP	FP	FP	✓	✓

Table 6.2: A comparison of Taser's four policies and RETRO against a set of scenarios used to evaluate Taser [20]. Taser's snapshot policy tracks all dependencies, NoI ignores IPC and signals, NoIAN also ignores file name and attributes, and NoIANC further ignores file content. FP indicates a false positive (undoing legitimate actions), FN indicates a false negative (missing parts of the attack), and ✓ indicates no false positives or negatives.

interesting ways.[1] In the first attack, the attacker changed the root password. In the second attack, the attacker downloaded and ran a Linux binary that scrubbed system log files of any mention of the attacker's login attempt.

For both of these attacks, RETRO was able to repair the system while preserving all legitimate user actions, as summarized in Table 6.1. In the password change attack, root was unable to log in after the attack, immediately exposing the compromise, although we still logged in as Alice and ran `texi2pdf`. In the second attack, all 59 repaired processes were from the attacker's log cleaning program, whose effects were undone.

For these real-world attacks, RETRO required minimal user input. RETRO required one piece of user input to repair the password change attack, because root's login attempt truly depended on root's entry in `/etc/passwd`, which was modified by the attacker.

[1] Most of the attackers simply ran a botnet binary or a port scanner.

Scenario	User input required
Illegal storage	None.
Content destruction	None. (Generates terminal diff compensating action.)
Unhappy student	None. (Generates terminal diff compensating action.)
Compromised database	None.
Software installation	Re-execute browser (or ignore browser state changes).
Inexperienced admin	Skip re-execution of attacker's login session.

Table 6.3: Required user input when repairing set of scenarios used to evaluate Taser.

In our experiment, the user told the network manager to ignore the conflict. RETRO required no user input for the log cleaning attack.

Synthetic attacks. To check if RETRO can recover from more insidious attacks, we constructed two synthetic attacks involving trojans; results for both are summarized in Table 6.1. For the first synthetic attack, we used the running example, where the attacker adds an account for `eve`, installs a botnet and a backdoor PHP script, and trojans the `/usr/bin/texi2pdf` shell script to restart the botnet. Legitimate users were unaware of this attack, and performed the same actions. Once the administrator detected the attack, RETRO reverted Eve's changes, including the `eve` account, the bot, and the trojan. As described in §4.2, RETRO used shepherded re-execution to undo the effects of the trojan without re-running the bulk of the trojaned application. As Table 6.1 indicates, RETRO re-executed several functions (`getpwnam`) to check if removing `eve`'s account affected any subsequent logins. One login session was affected—Eve's login—and RETRO's network manager required user input to confirm that Eve's login need not be re-executed.

One problem we discovered when repairing the running example attack is that the UID chosen for Alice by root's `useradd alice` command depends on whether `eve`'s account is present. If RETRO simply re-executed `useradd alice`, `useradd` would pick a different UID during re-execution, requiring RETRO to re-execute Alice's entire session. Instead, we made the `useradd` command part of the system library manager, so that during repair, it first tries to re-execute the action of adding user `alice` under the original UID, and only if that fails does it re-execute the full `useradd` program. This ensures that Alice's UID remains the same even after RETRO removes the `eve` account (as long as Alice's UID is still available).

Workload	No RETRO 1 core	RETRO 1 core	RETRO 2 cores	Log size	Snapshot size	# of objs	# of actions
Kernel build	295 sec	557 sec	351 sec	761 MB	308 MB	87,405	5,698,750
Web server	7260 req/s	3195 req/s	5453 req/s	98 MB	272 KB	508	185,315
HotCRP	20.4 req/s	15.1 req/s	20.0 req/s	81 MB	27 MB	19,969	939,418

Table 6.4: Performance and storage costs of RETRO for three workloads: building the Linux kernel, serving files as fast as possible using Apache [2] for 1 minute, and simulating requests to HotCRP [28] from the 30 minutes before the SOSP 2007 deadline, which averaged 2.1 requests per second [50] (running as fast as possible, this workload finished in 3–4 minutes). "# of objects" reflects the number of files, directory entries, and processes; not included are intermediate objects such as system call arguments. "# of actions" reflects the number of system call actions.

A second synthetic attack we tried was to trojan `/usr/sbin/sshd`. In this case, users were able to log in as usual, but undoing the attack required re-executing their login sessions with a good `sshd` binary. Because RETRO cannot rerun the remote ssh clients (and a new key exchange, resulting in different keys, makes TCP-level replay useless), RETRO's network manager asks the administrator to redo each ssh session manually. Of course, this would not be practical on a real system, and the administrator may instead resort to manually auditing the files affected by those login sessions, to verify whether they were affected by the attack in any way. However, we believe it is valuable for RETRO to identify all connections affected by the attack, so as to help the administrator locate potentially affected files. In practice, we hope that an intrusion detection system can notice such wide-reaching attacks; after a few user logins, the dependency graph indicates that unrelated user logins are all dependent on a previous login session, which an IDS may be able to flag.

Taser attacks. Finally, we compare RETRO to the state-of-the-art intrusion recovery system, Taser, under the attack scenarios that were used to originally evaluate Taser [20]. Table 6.3 summarizes the results.

In the first scenario, illegal storage, the attacker creates a new account for herself, stores illegal content on the system, and trojans the `ls` binary to mask the illegal content. RETRO rolls back the account, illegal files, and the trojaned `ls` binary, and uses the legitimate `ls` binary to re-execute all `ls` processes from the past. Even though the trojaned `ls` binary hid some files, the legitimate `ls` binary produces the same output,

because RETRO removes the hidden files during repair. As a result, there is no need to notify the user. If `ls`'s output did change, the terminal manager would have sent a diff to the affected users.

In the content destruction scenario, an attacker deletes a user's files. Once the user notices the problem, he uses RETRO to undo the attack. After recovering the files, RETRO generates a terminal output diff for the login session during which the user noticed the missing files (after repair, the user's `ls` command displays those files).

In the unhappy student scenario, a student exploits an `ftpd` bug to change permissions on a professor's grade file, then modifies the grade file in another login session, and finally a second accomplice user logs in and makes a copy of the grade file. In repairing the attack, RETRO rolls back the grade file and its permissions, re-executes the copy command (which now fails), and uses the terminal manager to generate a diff for the attackers' sessions, informing them that their copy command now failed.

In the compromised database scenario, an attacker breaks into a server, modifies some database records (in our case we used SQLite), and subsequently a legitimate user logs in and runs a script that updates database records of its own. RETRO rolls back the database file to a state before the attack, and re-executes the database update script to preserve subsequent changes, with no user input.

In the software installation scenario, the administrator installs the wrong browser plugin, and only detects this problem after running the browser and downloading some files. During repair, RETRO rolls back the incorrect plugin, and attempts to repair the browser using re-execution. Since RETRO encounters external dependencies in re-executing network applications, it requests the user to manually redo any interactions with the browser. In our experiment, the user ignored this external dependency, because he knew the browser made no changes to local state worth preserving.

In the inexperienced admin scenario, root selects a weak password for a user account, and an attacker guesses the password and logs in as the user. Undoing root's password change affects the attacker's login session, requiring one user input to confirm to the network manager that it's safe to discard the attacker's TCP connection.

In summary, RETRO correctly repairs all six attack scenarios posed by Taser, requiring

user input only in two cases: to re-execute the browser, and to confirm that it's safe to drop the attacker's login session. Taser requires application-specific policies to repair these attacks, and some attacks cannot be fully repaired under any policy. Taser's policies also open up the system to false negatives, allowing an adversary to bypass Taser altogether.

6.2 Technique effectiveness

In this subsection, we evaluate the effectiveness of RETRO's specific techniques, including re-execution, predicate checking, and refinement.

Re-execution is key to preserving legitimate user actions. As described in §6.1 and quantified in Table 6.1, RETRO re-executes several processes and functions to preserve and repair legitimate changes. Without re-execution, RETRO would have to conservatively roll back any files touched by the process in question, much like Taser's snapshot policy, which incurs false positives.

Without predicates, RETRO would have to perform conservative dependency propagation in the dependency graph. As in Taser, dependencies on attack actions quickly propagate to most objects in the graph, requiring re-execution of almost every process. This leads to re-execution of `sshd`, which requires user assistance. Table 6.1 shows that many of the objects repaired without predicates were not repaired with predicates enabled. Taser would roll back all of these objects (false positives). Thus, predicates are an important technique to minimize user input due to re-execution.

Without refinement of actor and data objects, RETRO would incur false dependencies via `/tmp` and `/etc/passwd`. As Table 6.1 shows, several functions (such as `getpwnam`) were re-executed in repairing from attacks. If RETRO was unable to re-execute just those functions, it would have re-executed processes like `sshd`, forcing the network manager to request user input. Thus, refinement is important to minimizing user input due to false dependencies.

6.3 Performance

We evaluate RETRO's performance costs in two ways. First, we consider costs of RETRO's logging during normal execution. To this end, we measure the CPU overhead and log size for several workloads. Table 6.4 summarizes the results. We ran our experiments on a 2.8GHz Intel Core i7 system with 8 GB RAM running a 64-bit Linux 2.6.35 kernel, with either one or two cores enabled.

The worst-case workload for RETRO is a system that uses 100% of CPU time and spends most of its time communicating between small processes. One such extreme workload is a system that continuously re-builds the Linux kernel; another example is an Apache server continuously serving small static files. For such systems, RETRO incurs a 89–127% CPU overhead using a single core, and generates about 100–150 GB of logs per day. A 2 TB disk ($100) can store two weeks of logs at this rate before having to garbage-collect older log entries. If a spare second core is available, and the application cannot take advantage of it, it can be used for logging, resulting in only 18–33% CPU overhead.

For a more realistic application, such as a HotCRP [28] paper submission web site, RETRO incurs much less overhead, since HotCRP's PHP code is relatively CPU-intensive. If we extrapolate the workload from the 30 minutes before the SOSP 2007 deadline [50] to an entire day, HotCRP would incur 35% CPU overhead on a single core (and almost no overhead if an additional unused core were available), and use about 4 GB of log space per day. We believe that these are reasonable costs to pay to be able to recover integrity after a compromise of a paper submission web site.

Second, we consider the time cost of repairing a system using RETRO after an attack. As Table 6.1 illustrated, RETRO is often effective at repairing only a small subset of objects and actions in the action history graph, and for attacks that affect the entire system state, such as the `sshd` trojan, user input dominates repair costs. To illustrate the costs of repairing a subset of the action history graph, we measure the time taken by RETRO to repair from a micro-benchmark attack, where the adversary adds an extraneous line to a log file, which is subsequently modified by a legitimate process. When only this

attack is present in RETRO's log (consisting of 10 process objects, 126 file objects, and 399 system call actions), repair takes 0.3 seconds. When this attack runs concurrently with a kernel build (as shown in Table 6.4), repair of the attack takes 4.7 seconds (10× longer), despite the fact that the log is 10,000× larger. This shows that RETRO's log indexing makes repair time depend largely on the number of affected objects, rather than the overall log size.

Chapter 7

Discussion

7.1 Limitations

An important assumption of RETRO is that the attacker does not compromise the kernel. Unfortunately, security vulnerabilities are periodically discovered in the Linux kernel [5, 6], making this assumption potentially dangerous. One solution may be to use virtual machine based techniques [16, 26], although it is difficult to distinguish kernel objects after a kernel compromise. We plan to explore ways of reducing trust in future work.

In our current prototype, if attackers compromise the kernel and obtain access to RETRO's log files, they may be able to extract sensitive information, such as user passwords or keys, that would not have been persistently stored on a system without RETRO. One possible solution may be to encrypt the log files and checkpoints, so that the administrator must reboot the system from a trusted CD and enter the password to initiate recovery.

Our current prototype can repair the effects of an attack only on a single machine, and relies on compensating actions to repair external state. As discussed in §1.6, WARP [11] overcomes this limit in the context of web applications, and AIRE [12] extends automated repair to distributed systems.

RETRO requires the system administrator to specify the initial intrusion point in order to undo the effects of the attack, and finding the initial intrusion point can be difficult. To overcome this situation, POIROT [25], as discussed in §1.6, can be used

as a companion tool for initiating recovery in RETRO, so given a security patch, it can retroactively recover from attacks that exploited vulnerabilities we have just fixed.

To run RETRO in a production environment, we should be able to perform multiple repairs, undoing a previous repair, repairing concurrently with normal execution. These operations translate into making additional checkpoints, and updating the graph accordingly after repair. For efficient re-execution, we also need to build more specialized repair managers, such as managers for a language runtime, a database, or an application like a web server or web browser. Finally, while RETRO's performance and storage overheads are already acceptable for some workloads, we need to further reduce them by not logging intermediate dependencies that can be reconstructed at repair time, or by transforming logs to a course-grained form.

7.2 Future directions

Auditable operating system. RETRO can be too aggressive in changing the system's state when repairing; we believe this blocks the adoption of our system in practice. In the future, we want to explore the design of a new operating system, focusing solely on auditing of past executions: e.g., users can identify when the trojan was installed by running trojan detectors against the entire history. We need two kinds of system components to achieve system-wide auditing: a *versioning filesystem* to trace the history of file modifications, and *time-travel invocations* to run programs at a particular time in history.

The *versioning filesystem* preserves every modification of files and never overwrites existing files. On top of the versioning filesystem, we will implement *time-travel invocations* with which users can go back to the history and run a program against it. Combining both mechanisms together, we can efficiently audit the past execution of an operating system. For example, if a program suddenly starts crashing, a user can attempt to run this program on every moment in history (doing binary search within the given time period), so the user can identify what change in the past made the program crash.

Interactive recovery. RETRO currently repairs the system's state by modifying its history and makes those changes immediately visible at the present filesystem. For end users, this repair process seems opaque in understanding the real effects of replaying forward after fixing the history. To solve this problem, we plan to represent low-level re-execution details, such as exploring and fixing the system's state, in a user-friendly form, perhaps as a scripting language. By doing so, users can manipulate a recovery plan, the script, before initiating the actual recovery. Also, we plan to maintain the history immutable so replaying does not affect anything until a user explicitly merges the repaired state to the present filesystem.

Representing privileges in dependency graphs. Existing recovery mechanisms requires a special privilege for repairing, because the current dependency graphs do not embed any privilege model of each operation. However, we plan to make undo and redo mechanism available to normal users, not only for repairing intrusion but also for recovering from their own mistakes. But, at the same time, we don't want to break the existing privilege model in commodity operating systems, nor leak user's private data while processing repair requests of adversarial users. Whenever required, we should ask users to provide a password for the necessary credential; for example, the recovery script can express a change of a credential as a sudo command.

User interface. Which parts of the dependency graph are interesting to look up? And how to query and navigate history? These are the important questions to answer in terms of usability and practicality. One possible way to visualize history is to memorize rendezvous actions that users might remember for requesting a recovery.

For security concerns, users should be able to invoke the interface via a safe channel; for example, after a user mistakenly executes `rm /`, the user should be able to restore the whole system state. To do so, we plan to design an interface, similar to Linux's Magic SysRq, so users can safely restore to the previous system's state, at any time.

Chapter 8

Related work

This section relates RETRO to industrial and academic solutions for recovery after a compromise, and prior techniques that RETRO builds on.

8.1 Recovery

One line of industrial solutions is anti-virus tools, which can revert changes made by common malware, such as Windows registry keys and files comprising a known virus. For example, tools such as [39] can generate remediation procedures for a given piece of malware. While such techniques work for known malware that behaves in predictable ways, they incur both false positives and false negatives, especially for new or unpredictable malware, and may not be able to recover from attacks where some information is lost, such as file deletions or overwrites. They also cannot repair changes that were a side-effect of the attack, such as changes made by a trojaned program, or changes made by an interactive adversary, whereas RETRO can undo such changes.

Another line of industrial solutions is systems that help users roll back unwanted changes to system state. These solutions include Windows System Restore [22], Windows Driver Rollback [35], Time Machine [4], and numerous backup tools. These tools perform coarse-grained recovery, and require the user to identify what files were affected. RETRO uses the action history graph to track down *all* effects of an attack, repairs *precisely* those changes, and repairs all *side-effects* of the attack, without requiring the user to guess

what files were affected.

A final line of popular solutions is using virtual machines as a form of whole-system backup. Using ReVirt [16] or Moka5 [10, 36], an administrator can roll back to a checkpoint before an attack, losing both the attacker's changes and any legitimate changes since that point. One could imagine a system that replays recorded legitimate network packets to the virtual machine to re-apply legitimate changes. However, if there are even subtle dependencies between omitted and replayed packets, the replayed packets will result in conflicts or external dependencies, requiring user input to proceed. By recording dependencies and re-executing actions at many levels of abstraction using refinement, RETRO avoids such conflicts and can preserve legitimate changes without user input.

Academic research has tried to improve over the industrial solutions by attempting to make solutions more automatic. Brown's undoable email store [18, 40**?**] shows how an email server can recover from operator mistakes, by turning all operations into *verbs*, such as SMTP or IMAP commands. Unlike RETRO, Brown's approach is limited to recovering from accidental operator mistakes. As a result, it cannot deal with an adversary that goes outside of the *verb* model and takes advantage of a vulnerability in the IMAP server software, or guesses root's password to log in via ssh. Moreover, it cannot recover from actions that had system-wide effects spanning multiple applications, files, and processes.

The closest related work to RETRO is Taser [20], which uses taint tracking to find files affected by a past attack. Taser suffers from false positives, erroneously rolling back hundreds or thousands of files. To prevent false positives, Taser uses a white-list to ignore taint for some nodes or edges. This causes false negatives, so an attacker can bypass Taser altogether. While extensions of Taser catch some classes of attacks missed due to false negatives [46], RETRO has no need for white-listing. RETRO recovers from all attacks presented in the Taser paper with no false positives or false negatives. RETRO avoids Taser's limitations by using a design based on the action history graph, and techniques such as predicates and re-execution, as opposed to Taser's taint propagation.

Polygraph [34] uses taint tracking to recover from compromised devices in a data

replication system, and incurs false positives like Taser. Unlike RETRO, Polygraph can recover from compromises in a distributed system.

8.2 Retroactive auditing

Retroactive auditing is a way to detect past intrusions by using security patches. Its core idea and techniques are similar to RETRO in that both are using record-and-replay. However, unlike RETRO, which focuses on repairing from the identified attacks, auditing systems are focusing on how to identify the past intrusions. The idea of using security patches is originally proposed by RAD [48], in the context of unmodified binaries. WARP [11] extended this idea further to web applications; given a security patch, WARP detects the past intrusions exploiting the vulnerability that the security patch fixed, and repairs from the entry points of identified attacks.

However, RAD and WARP require too much re-execution; RAD reruns two times for unmodified and patched binaries for comparing outputs of them, and WARP reruns all past requests that executed files modified by the patch. POIROT [25] attempts to solve this particular problem, performance of re-execution, by avoiding redundant executions while replaying with memorizing their intermediate results of executions. RAIL [14] makes this approach even more promising by requiring developers to use explicit APIs for correct replaying.

8.3 Other techniques

The use of dependency information for security has been widely explored in many contexts, including information flow control [30, 51], taint tracking [50], data provenance [9], forensics [26], system integrity [8], and so on. A key difference in RETRO's action history graph is the use of exact dependency data to decide whether a dependency has semantically changed at repair time.

RETRO assumes that intrusion detection and analysis tools, such as [7, 13, 16, 17, 23, 24, 26, 27, 29, 46, 49], detect attacks and pinpoint attack edges. RETRO's intrusion

detection is based on BackTracker [26]. A difference is that RETRO's action history graph records more information than BackTracker, which RETRO needs for repair (but doesn't use yet for detection).

Transactions [38, 42] help revert unwanted changes before commit, whereas RETRO can selectively undo "committed" actions. Database systems use compensating transactions to revert committed transactions, including malicious transactions [3, 32]; RETRO similarly uses compensating actions to deal with externally-visible changes.

Chapter 9

Conclusion

This dissertation explored an automatic way of recovering from intrusions by using system-wide history. To demonstrate our idea and techniques, we designed, implemented and evaluated RETRO, an automatic intrusion recovery system in the context of operating systems. To repair system integrity from past attacks, RETRO uses an action history graph to track complete system-wide dependencies. Given the action history graph, RETRO rolls back objects directly affected by the attack, and re-executes legitimate actions indirectly affected by our changes in the history. To minimize user's input, RETRO avoids re-execution whenever possible, and provides compensating actions for external dependencies. RETRO's key techniques for minimizing re-execution include predicates to check semantic equivalence of actions; refinement to embed high-level semantics of applications; and shepherded re-execution to enable fine-grained re-execution. A prototype of RETRO for Linux recovers from a mix of ten real-world and synthetic attacks, repairing all side-effects of the attack in all cases. Six attacks required no user input to repair, and RETRO required significant user input in only two cases involving trojaned network-facing applications. In the near future, we believe the idea of RETRO, automatic intrusion recovery with system-wide history, may become even more feasible with the advances of computing and storage technologies.

Bibliography

[1] The Honeynet Project. http://www.honeynet.org/.

[2] Apache web server, May 2010. http://httpd.apache.org/.

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

[4] Apple Inc. What is Mac OS X - Time Machine. http://www.apple.com/macosx/what-is-macosx/time-machine.html.

[5] J. Arnold and M. F. Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the ACM EuroSys Conference*, Nuremberg, Germany, Mar. 2009.

[6] J. Arnold, T. Abbott, W. Daher, G. Price, N. Elhage, G. Thomas, and A. Kaseorg. Security impact ratings considered harmful. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, Monte Verita, Switzerland, May 2009.

[7] AVG Technologies. Why traditional anti-malware solutions are no longer enough. http://download.avg.com/filedir/other/pf_wp-90_A4_us_z3162_20091112.pdf, October 2009.

[8] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, MITRE Corp., Bedford, MA, Apr. 1977.

[9] U. Braun, A. Shinnar, and M. Seltzer. Securing provenance. In *Proceedings of the 3rd Usenix Workshop on Hot Topics in Security*, San Jose, CA, July 2008.

[10] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam. The Collective: A cache-based system management architecture. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, pages 259–272, Boston,

MA, May 2005.

[11] R. Chandra, T. Kim, M. Shah, N. Narula, and N. Zeldovich. Intrusion recovery for database-backed web applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 101–114, Cascais, Portugal, Oct. 2011.

[12] R. Chandra, T. Kim, and N. Zeldovich. Asynchronous repair for distributed web services. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, Nov. 2013.

[13] CheckPoint, Inc. IPS-1 intrusion detection and prevention system. http://www.checkpoint.com/products/ips-1/.

[14] H. Chen, T. Kim, X. Wang, M. F. Kaashoek, and N. Zeldovich. Identifying information disclosure in web applications with retroactive auditing. In submission.

[15] J. Corbet. A checkpoint/restart update. http://lwn.net/Articles/375855/, February 2010.

[16] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224, Boston, MA, Dec. 2002.

[17] S. Forrest, S. Hofmeyr, and A. Somayaji. The evolution of system-call monitoring. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, pages 418–430, December 2008.

[18] A. Fox. Toward recovery-oriented computing. In *In Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, pages 1–3, 2002.

[19] FreeBSD. What is securelevel? http://www.freebsd.org/doc/en_US.ISO8859-1/books/faq/security.html#SECURELEVEL.

[20] A. Goel, K. Po, K. Farhadi, Z. Li, and E. D. Lara. The Taser intrusion recovery system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 163–176, Brighton, UK, Oct. 2005.

[21] D. Goodin. Kernel.org linux repository rooted in hack attack, August 2011. http:

//www.theregister.co.uk/2011/08/31/linux_kernel_security_breach.

[22] B. Harder. Microsoft Windows XP system restore. http://msdn.microsoft.com/en-us/library/ms997627.aspx, April 2001.

[23] A. Joshi, S. King, G. Dunlap, and P. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 91–104, Brighton, UK, Oct. 2005.

[24] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 18–29, Fairfax, VA, Nov. 1994.

[25] T. Kim, R. Chandra, and N. Zeldovich. Efficient patch-based auditing for web application vulnerabilities. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 193–206, Hollywood, CA, Oct. 2012.

[26] S. T. King and P. M. Chen. Backtracking intrusions. *ACM Transactions on Computer Systems*, 23(1):51–76, Feb. 2005.

[27] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen. Enriching intrusion alerts through multi-host causality. In *Proceedings of the 12th NDSS*, San Diego, CA, February 2005.

[28] E. Kohler. Hot crap! In *Proceedings of the Workshop on Organizing Workshops, Conferences, and Symposia for Computer Systems*, San Francisco, CA, Apr. 2008.

[29] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th Usenix Security Symposium*, Montreal, Canada, August 2009.

[30] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 321–334, Stevenson, WA, Oct. 2007.

[31] A. Lewis. LVM HOWTO: Snapshots. http://www.tldp.org/HOWTO/LVM-HOWTO/

snapshotintro.html.

[32] P. Liu, P. Ammann, and S. Jajodia. Rewriting histories: Recovering from malicious transactions. *Journal of Distributed and Parallel Databases*, 8(1):7–40, 2000.

[33] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 29–40, June 2001. FREENIX track.

[34] P. Mahajan, R. Kotla, C. C. Marshall, V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, and T. Wobber. Effective and efficient compromise recovery for weakly consistent replication. In *Proceedings of the ACM EuroSys Conference*, pages 131–144, Nuremberg, Germany, March 2009.

[35] Microsoft. How to use the roll back driver feature in Windows XP. http://support.microsoft.com/kb/283657, August 2007.

[36] MokaFive, Inc. Mokafive, virtual desktops for businesses and personal use. http://www.mokafive.com/.

[37] NetApp. Snapshot. http://www.netapp.com/us/products/platform-os/snapshot.html.

[38] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, Brighton, UK, Oct. 2005.

[39] R. Paleari, L. Martignoni, E. Passerini, D. Davidson, M. Fredrikson, J. Giffin, , and S. Jha. Automatic generation of remediation procedures for malware infections. In *Proceedings of the 19th Usenix Security Symposium*, Washington, DC, Aug. 2010.

[40] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery oriented computing (roc): Motivation, definition, techniques,. Technical report, Berkeley, CA, USA, 2002.

[41] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Proceedings of the 1995 USENIX Annual Technical Conference*, pages

213–223, New Orleans, LA, Jan. 1995.

[42] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating systems transactions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 161–176, Big Sky, MT, Oct. 2009.

[43] O. Rodeh. B-trees, shadowing, and clones. *ACM Transactions on Storage*, 3(4): 1–27, 2008.

[44] M. Satyanarayanan. Scalable, secure and highly available file access in a distributed workstation environment. *IEEE Computer*, 23(5):9–21, May 1990.

[45] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, Oct. 2007.

[46] F. Shafique, K. Po, and A. Goel. Correlating multi-session attacks via replay. In *Proceedings of the Second Workshop on Hot Topics in System Dependability*, Seattle, WA, November 2006.

[47] B. Spengler. grsecurity. http://www.grsecurity.net/.

[48] X. Wang, N. Zeldovich, and M. F. Kaashoek. Retroactive auditing. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, Shanghai, China, July 2011. 5 pages.

[49] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM CCS*, Alexandria, VA, October-November 2007.

[50] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 291–304, Big Sky, MT, Oct. 2009.

[51] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 263–278, Seattle, WA, Nov. 2006.