Intrusion Recovery Using Selective Re-execution

Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek
MIT CSAIL

ABSTRACT

RETRO repairs a desktop or server 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*, which is a detailed dependency graph describing the system's execution. RETRO uses *refinement* to describe graph objects and actions at multiple levels of abstraction, which allows for precise dependencies. 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 user involvement and re-execution, RETRO uses *predicates* to *selectively re-execute* only actions that were semantically affected by the adversary's changes, 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 often repair the system without user involvement, and avoids false positives and negatives from previous solutions. 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.

1 INTRODUCTION

Despite our best efforts to build secure computer systems, intrusions are nearly unavoidable in practice. When faced with an intrusion, a user is typically forced to reinstall their system from scratch, and to manually recover any documents and settings they might have had. Even if the user diligently makes a complete backup of their system every day, recovering from the attack requires rolling back to the most recent backup before the attack, thereby losing any changes made since then. Since many adversaries go to great lengths to prevent the compromise from being discovered, it can take days or weeks for a user to discover that their machine has been broken into, resulting in a loss of all user work from that period of time.

This paper presents RETRO, a system for retroactively undoing past attacks and their indirect effects on a single machine. With RETRO, an administrator specifies offending actions from the past, such as a TCP connection or an HTTP request from an adversary, that they want to undo. RETRO then repairs the system's state (the file system) by selectively undoing the offending actions—that is, constructing a new system state, as if the offending actions never took place, but all legitimate actions remained. Thus, by selectively undoing the adversary's changes while preserving user data, RETRO makes intrusion recovery more practical.

To illustrate the challenges facing RETRO, consider the following attack, which we will use as a running example in this paper. 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 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.

As described in §2, most previous systems 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 [17]) 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).

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 four 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 those objects (corresponding to actions such as a process reading a file). 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. The action history graph also captures the semantics of each dependency (e.g., the arguments and return values of an action).

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 uses *predicates* to do *selective re-execution* of just the actions whose dependencies are semantically different after repair, thereby minimizing cascading re-execution. For example, if Eve 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.

Finally, to selectively re-execute existing applications, RETRO uses *shepherded re-execution* to monitor the re-execution of processes (§5.2.3), and stops re-execution when the process state converges to the original execution (such as when a process issues an identical `exec` call).

Using a prototype of RETRO for Linux, we show that RETRO can recover from both real-world and synthetic attacks, including our running example, 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, shepherded re-execution, 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 [23] 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.

The rest of the paper is organized as follows. The next section compares RETRO with related work. §3 presents an overview of RETRO's architecture and workflow. §4 discusses RETRO's action history graph in detail, and §5 describes RETRO's repair managers. Our prototype implementation is described in §6, and §7 evaluates the effectiveness and performance of RETRO. Finally, §8 discusses the limitations and future work, and §9 concludes.

2 RELATED WORK

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

2.1 Repair solutions

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 [34] 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 [18], Windows Driver Rollback [30], 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 [14] or Moka5 [11, 31], 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

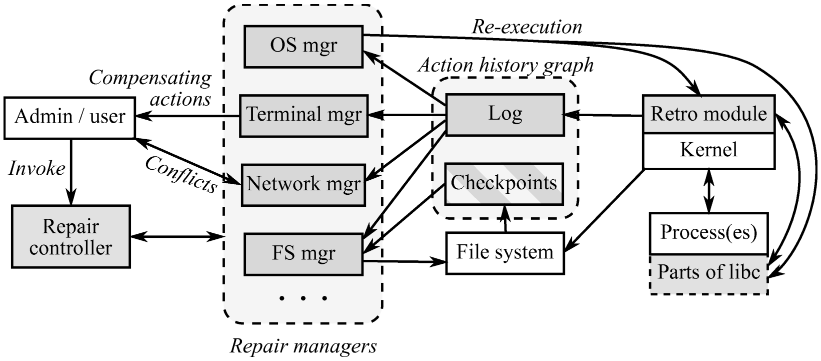

Figure 1: Overview of RETRO's architecture, including major components and their interactions. Shading indicates components introduced by RETRO. Striped shading of checkpoints indicates that RETRO reuses existing file system snapshots when available.

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 [10] 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 [17], 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 [40], 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 [29] 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.

2.2 Related techniques

The use of dependency information for security has been widely explored in many contexts, including information flow control [25, 45], taint tracking [44], data provenance [9], forensics [21], 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, 12, 14, 15, 19–22, 24, 40, 43], detect attacks and pinpoint attack edges. RETRO's intrusion detection is based on BackTracker [21]. 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 [33, 36] 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, 27]; RETRO similarly uses compensating actions to deal with externally-visible changes.

3 OVERVIEW

RETRO consists of several components, as shown in Figure 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, and outlines the assumptions made by RETRO about the system and the adversary.

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 [26, 32, 37, 38], 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 2.

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 [23] 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.

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 [21] 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 paper, we assume that an intrusion detection system exists, and we do not describe our BackTracker-like tool in any more detail.

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 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.

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

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

[2] Each object and action in the action history graph has a unique name, as described in §5.

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.

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, for this paper, 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 [16, 28, 39, 41], to achieve this goal in practice.

4 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.

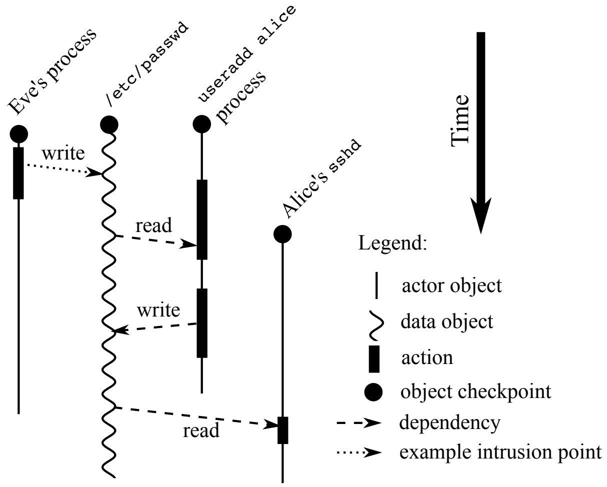

Figure 2: A simplified view of the action history graph depicting Eve's attack in our running example. In this graph, attacker Eve adds an account for herself to /etc/passwd, after which root adds an account for Alice, and Alice logs in via ssh. As an example, we consider Eve's write to the password 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 §4.3 and Figure 4.

4.1 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 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

Function or variable		Semantics
$set\langle checkpt\rangle$	object.checkpts	Set of available checkpoints for this object.
$void$	object.rollback(c)	Roll back this object to checkpoint c.
$set\langle action\rangle$	actor_object.actions	Set of actions that comprise this actor object.
$set\langle action\rangle$	data_object.readers	Set of actions that have a dependency from this data object.
$set\langle action\rangle$	data_object.writers	Set of actions that have a dependency to this data object.
$set\langle data\_object\rangle$	data_object.parts	Set of data objects whose state is part of this data object.
$actor\_object$	action.actor	Actor containing this action.
$set\langle data\_object\rangle$	action.inputs	Set of data objects that this action depends on.
$set\langle data\_object\rangle$	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.

Figure 3: Object (top) and action (bottom) repair manager API.

file, as in Figure 2, 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 2, 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 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 2, 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 §4.4 and Figure 5).

4.2 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 Figure 3.

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 2. 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 §5.

An object manager provides one main procedure invoked during repair, $o.rollback(v)$, 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 §5.2.3; 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, $a.\mathit{redo}()$ 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 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, $a.\mathit{equiv}()$ 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, $a.\mathit{connect}()$ 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).

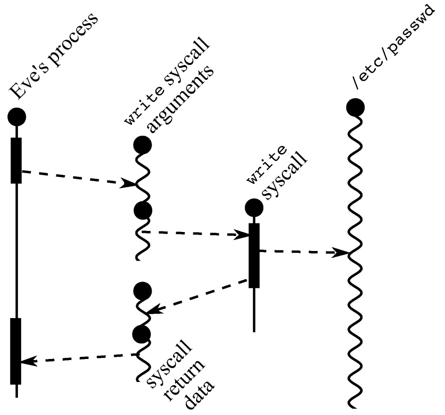

Figure 4: An illustration of the system call actor object and arguments and return value data objects, for Eve's write to the password file from Figure 2. Legend is the same as in Figure 2.

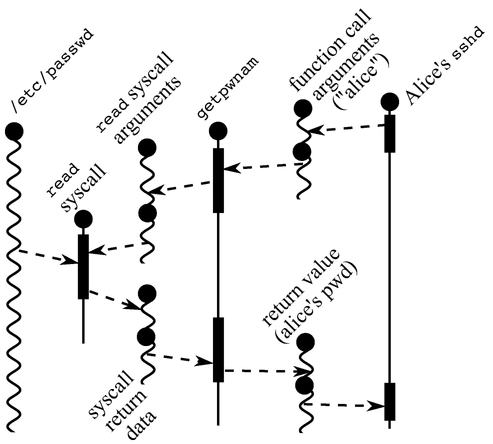

Figure 5: An illustration of refinement in an action history graph, depicting the use of additional actors to represent a re-executable call to `getpwnam` from `sshd`. Legend is the same as in Figure 2.

4.3 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 2, 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 4. 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 5 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. §5 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 6 and §4.5.

4.4 Refining data objects: Finer-grained data 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. 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 $o.parts$ property, as shown in Figure 3. In most cases, the manager tracks its parts through hierarchical names, as we discuss in §5.

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 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.

4.5 Repair controller

RETRO uses a *repair controller* to repair system state with the help of object and action managers. Figure 6 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.

```
function ROLLBACK($node$, $checkpt$)
    $node$.rollback($checkpt$)
    $state[node] := checkpt$

function PREPAREREDO($action$)
    if $\neg action$.connect() then return FALSE
    if $state[action.\text{actor}] > action$ then
        $cps := action.\text{actor.checkpts}$
        $cp := \max(c \in cps \mid c \leq action)$
        ROLLBACK($action$.actor, $cp$)
        return FALSE
    for all $o \in (action.\text{inputs} \cup action.\text{outputs})$ do
        if $state[o] \leq action$ then continue
        ROLLBACK($o, \max(c \in o.\text{checkpts} \mid c \leq action)$)
        return FALSE
    return TRUE

function PICKACTION()
    $actions := \emptyset$
    for all $o \in state \mid o$ is actor object do
        $actions$ += $\min(a \in o.\text{actions} \mid a > state[o])$
    for all $o \in state \mid o$ is data object do
        $actions$ += $\min(a \in o.\text{readers} \cup$
                                    $o.\text{writers} \mid a > state[o])$
    return min(actions)

function REPAIRLOOP()
    while $a :=$ PICKACTION() do
        if $a$.equiv() and $state[o] \geq a$,
          $\forall o \in a.\text{outputs} \cup a.\text{actor}$ then
            for all $i \in a.\text{inputs} \cap \text{keys}(state)$ do
                $state[i] := a$
            continue    ▷ skip semantically-equivalent action
        if PREPAREREDO($a$) then
            $a$.redo()
            for all $o \in a.\text{inputs} \cup a.\text{outputs} \cup a.\text{actor}$ do
                $state[o] := a$

function REPAIR($repair\_obj$, $repair\_cp$)
    ROLLBACK($repair\_obj$, $repair\_cp$)
    REPAIRLOOP( )
```

Figure 6: The repair algorithm.

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 §4.3.

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 §4.4.

5 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.

5.1 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 [26, 32, 37, 38], 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 §5.3).

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.

5.2 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.

5.2.1 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 [13, 35], our OS manager checkpoints actors only in their "initial" state immediately prior to `exec`, denoted by $\bot$. 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.

5.2.2 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 4. 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 4, 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.

5.2.3 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 $p.rollback(\bot)$, 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 4). 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.

5.3 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 $\langle \textit{device}, \textit{part}, \textit{inode}, \textit{name} \rangle$. 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 $\bot$ 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.

5.4 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 5 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 5, 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.

5.5 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.

5.6 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.

6 Implementation

We implemented a prototype of Retro for Linux,[3] components of which are summarized in Figure 7. 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 [37]. 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 completeness. Third, Retro compresses the resulting log files to save space.

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

Figure 7: Components of our Retro prototype, and an estimate of their complexity, in terms of lines of code.

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

Figure 8: 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 §7.

7 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?

7.1 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 [17].

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 interesting ways.[4] In the first attack, the attacker changed the root password. In the second attack, the attacker downloaded and ran a Linux

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

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

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

Figure 9: A comparison of Taser's four policies and RETRO against a set of scenarios used to evaluate Taser [17]. 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.

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 Figure 8. 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. 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 Figure 8. 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 §5.2.3, RETRO used shepherded re-execution to undo the effects of the trojan without re-running the bulk of the trojaned application. As Figure 8 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).

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 [17]. Figure 9 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

Workload	Without RETRO	With RETRO		Log size	Snapshot size	# of objects	# of actions
	1 core	1 core	2 cores				
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

Figure 10: 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 [23] from the 30 minutes before the SOSP 2007 deadline, which averaged 2.1 requests per second [44] (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.

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.

7.2 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 §7.1 and quantified in Figure 8, 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. Figure 8 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 Figure 8 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.

7.3 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. Figure 10 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 [23] 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 [44] 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 Figure 8 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 Figure 10), 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.

8 DISCUSSION AND FUTURE WORK

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 [14, 21], 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 only repair the effects of an attack on a single machine, and relies on compensating actions to repair external state. In future work, we plan to explore ways to extend automated repair to distributed systems, perhaps based on the ideas from [29, 42].

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. In future work, we hope to leverage the extensive data available in RETRO's dependency graph to build intrusion detection tools that can better pin-point intrusions. Alternatively, instead of trying to pinpoint the attack, we may be able to use RETRO to retroactively apply security patches into the past, and re-execute any affected computations, thus eliminating any attacks that exploited the vulnerability in question.

We did not have space to address several practical aspects of using RETRO, such as performing multiple repairs or undoing a repair. These operations translate into making additional checkpoints, and updating the graph accordingly after repair. Also, as hinted at in §5, we plan to explore the use of 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 plan to further reduce them by not logging intermediate dependencies that can be reconstructed at repair time.

9 CONCLUSION

RETRO repairs system integrity from past attacks by using an action history graph to track system-wide dependencies, roll back affected objects, and re-execute legitimate actions affected by the attack. RETRO minimizes user input by avoiding re-execution whenever possible, and by using compensating actions for external dependencies. RETRO's key techniques for minimizing re-execution include predicates, refinement, and shepherded 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.

ACKNOWLEDGMENTS

We thank Victor Costan, Robert Morris, Jacob Strauss, the anonymous reviewers, and our shepherd, Adrian Perrig, for their feedback. Quanta Computer partially supported this work. Taesoo Kim is partially supported by the Samsung Scholarship Foundation, and Nickolai Zeldovich is partially supported by a Sloan Fellowship.

References

[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 *Proc. 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 *Proc. 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, Oct 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 *Proc. of the 3rd Usenix Workshop on Hot Topics in Security*, San Jose, CA, Jul 2008.

[10] A. B. Brown and D. A. Patterson. Undo for operators: Building an undoable e-mail store. In *Proc. of the 2003 Usenix ATC*, pages 1–14, San Antonio, TX, Jun 2003.

[11] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. Lam. The Collective: A cache-based system management architecture. In *Proc. of the 2nd NSDI*, pages 259–272, Boston, MA, May 2005.

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

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

[14] 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 *Proc. of the 5th OSDI*, pages 211–224, Boston, MA, Dec 2002.

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

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

[17] A. Goel, K. Po, K. Farhadi, Z. Li, and E. D. Lara. The Taser intrusion recovery system. In *Proc. of the 20th ACM SOSP*, pages 163–176, Brighton, UK, Oct 2005.

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

[19] A. Joshi, S. King, G. Dunlap, and P. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proc. of the 20th ACM SOSP*, pages 91–104, Brighton, UK, Oct 2005.

[20] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *Proc. of the 2nd ACM CCS*, pages 18–29, Fairfax, VA, Nov 1994.

[21] S. T. King and P. M. Chen. Backtracking intrusions. *ACM TOCS*, 23(1):51–76, Feb 2005.

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

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

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

[25] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proc. of the 21st ACM SOSP*, pages 321–334, Stevenson, WA, Oct 2007.

[26] A. Lewis. LVM HOWTO: Snapshots. http://www.tldp.org/HOWTO/LVM-HOWTO/snapshotintro.html.

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

[28] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proc. of the 2001 Usenix ATC*, pages 29–40, Jun 2001. Freenix track.

[29] 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 *Proc. of the ACM EuroSys Conference*, pages 131–144, Nuremberg, Germany, Mar 2009.

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

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

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

[33] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *Proc. of the 20th ACM SOSP*, Brighton, UK, Oct 2005.

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

[35] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Proc. of the 1995 Usenix ATC*, pages 213–223, New Orleans, LA, Jan. 1995.

[36] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating systems transactions. In *Proc. of the 22nd ACM SOSP*, pages 161–176, Big Sky, MT, Oct 2009.

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

[38] M. Satyanarayanan. Scalable, secure and highly available file access in a distributed workstation environment. *IEEE Computer*, pages 9–21, May 1990.

[39] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proc. of the 21st ACM SOSP*, Stevenson, WA, Oct 2007.

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

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

[42] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proc. of the 14th NDSS*, San Diego, CA, Feb-Mar 2007.

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

[44] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proc. of the 22nd ACM SOSP*, pages 291–304, Big Sky, MT, Oct 2009.

[45] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. of the 7th OSDI*, pages 263–278, Seattle, WA, Nov 2006.