The Taser Intrusion Recovery System

Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li*, Eyal de Lara*
Dept. of Electrical and Computer Engineering, Dept. of Computer Science*
University of Toronto

ABSTRACT

Recovery from intrusions is typically a very time-consuming operation in current systems. At a time when the cost of human resources dominates the cost of computing resources, we argue that next generation systems should be built with automated intrusion recovery as a primary goal. In this paper, we describe the design of Taser, a system that helps in selectively recovering legitimate file-system data after an attack or local damage occurs. Taser reverts tainted, i.e. attack-dependent, file-system operations but preserves legitimate operations. This process is difficult for two reasons. First, the set of tainted operations is not known precisely. Second, the recovery process can cause conflicts when legitimate operations depend on tainted operations. Taser provides several analysis policies that aid in determining the set of tainted operations. To handle conflicts, Taser uses automated resolution policies that isolate the tainted operations. Our evaluation shows that Taser is effective in recovering from a wide range of intrusions as well as damage caused by system management errors.

Categories and Subject Descriptors

E.5 [**Files**]: Backup/recovery; D.4.5 [**Reliability**]: Backup procedures, Fault-tolerance; D.4.6 [**Security and Protection**]: Information flow controls, Invasive software (e.g., viruses, worms, Trojan horses); K.6.5 [**Security and Protection**]: Unauthorized access (e.g., hacking, phreaking)

General Terms

Management, Reliability, Security

Keywords

File Systems, Intrusion Analysis, Intrusion Recovery, Snapshots

1. INTRODUCTION

When systems are compromised, one of the most error-prone and time-consuming tasks is recovery of persistent data. Recovery is performed after an attack is discovered and typically involves many steps: installation of a new system image that includes the operating system and all applications, installation of software patches that fix known vulnerabilities, and retrieval of uncorrupted user data. Each of these recovery steps is manual, tedious and time-intensive.

Today, snapshot-based file-systems [22, 25] provide a well understood and commonly deployed recovery solution [30]. This method gets rid of all corrupted data, but unfortunately, it also gets rid of useful data not related to the intrusion, and this data must then be manually retrieved or recovered separately.

Given the high human costs associated with recovery, sacrificing some machine and networking resources for automating the recovery process should be an attractive proposition. Moreover, with the rapid and continuous decline in computing, networking, and storage costs, logging *all* system activity, which is needed for recovery, is now technically and economically feasible [26, 5, 12, 8].

This paper describes the design of Taser, a system that *selectively* recovers file-system data after an intrusion is detected or after some damage is caused by system management error. Taser reverts the effects of file-system modification operations of *tainted* processes, or processes that were affected by the intrusion, but preserves the operations of legitimate processes. This approach has similarities with system software upgrade where the upgrade can be selectively rolled back without affecting the rest of the system [1, 10].

The Taser recovery method raises two main challenges. First, the set of file-system operations that depend on an intrusion, or tainted operations, is not known precisely because legitimate user activities may unknowingly interact with attack-dependent objects. For example, a legitimate process may become falsely dependent on a tainted process as a result of reading a shared file such as a log file or a password file. Second, legitimate operations that need to be preserved may depend on tainted operations that should be reverted. For example, a legitimate operation could have modified a file that was created by a tainted operation. If the tainted create operation is reverted by simply removing the file then the legitimate modification is lost. With intrusion recovery, the challenge is to preserve all legitimate file-system operations, revert or isolate the ill-effects of tainted operations, and at the same time, automate recovery as much as possible.

Taser operates in three steps: auditing, analysis and recovery. The auditing step uses the Forensix system to securely and accurately audit operations related to files, processes and sockets [8]. Forensix attributes file-system operations to processes and allows replaying them. The analysis step uses the audit information to taint processes, files and sockets based on dependency rules. For example, a rule may taint a process that reads a tainted file. Together with an initial set of externally provided tainted processes, files or sockets, these rules help separate the tainted from the legitimate file-system operations. Finally, the recovery step reverts the effects of tainted operations by selectively replaying only the legitimate operations on tainted file-system objects.

The choice of dependency rules involves an inherent trade-off. These rules can be chosen conservatively. For example, any interaction between processes, files and sockets could be used to taint an object. This approach simplifies the recovery process, but it can mistakenly mark legitimate operations as tainted, so data is lost. In contrast, some of the interactions could be ignored, which reduces the number of false dependencies. However, this approach can miss tainted operations and recovery becomes more complicated due to conflicts that can arise between tainted and legitimate operations. The analysis step exposes this trade-off by providing a choice of

SOSP'05, October 23–26, 2005, Brighton, United Kingdom.

policies from conservative to optimistic. The optimistic policies ignore certain dependencies. For example, a policy may optimistically assume that reading certain log files does not cause a process to be tainted.

The optimistic policies can lead to conflicts during recovery. Conflicts arise when legitimate operations depend on tainted operations. For example, a legitimate file may have been created in a tainted directory. Simply removing the tainted directory conflicts with the legitimate file that needs to be preserved. Such conflicts become more likely when the intrusion or administration error is detected long after its occurrence. Since manual resolution of conflicts is an arduous and error-prone process, Taser uses automatic conflict resolution methods during recovery. To do so, it separates file-system operations into name, content and attribute operations. This approach simplifies resolution, allows recovery actions that are suited for each type of operation, and enables fully automatic name and attribute conflict resolution.

This paper shows that the Taser intrusion recovery system simplifies the task of recovering file-system data after intrusions or damage caused by system management error. Taser implements several analysis policies that help in deriving tainted operations and it provides automated resolution methods for tainted operations that conflict with legitimate user operations. Our evaluation shows that Taser correctly recovers from a wide range of intrusions as well as erroneous system management activities.

The rest of the paper provides the details of our approach. Section 2 presents the design of the Taser recovery system. Section 3 describes enhancements that improve the accuracy of the recovery process. Section 4 discusses key implementation issues. Section 5 provides a detailed evaluation of our system. Section 6 discusses related work in the area. Finally, Section 7 presents our conclusions and directions for future work.

2. DESIGN OF THE TASER SYSTEM

The Taser system recovers file-system data after an intrusion or management error by reverting the file-system modification operations affected by the intrusion while preserving the modifications made by legitimate processes. In the rest of this paper, we use the term intrusion to mean a system compromise as well as a management error.

The Taser architecture consists of three main components: auditor, analyzer, and resolver. The auditor runs in the background during normal system operation and creates an audit log of all system activities including file-system operations. The analyzer and resolver are executed by an administrator during the recovery process. Recovery is started after an intrusion has been detected externally such as by an intrusion detection system (IDS) or by an administrator. The analyzer uses the audit log to determine the set of *tainted* file-system objects that were affected by the intrusion. The resolver uses this set of tainted objects and the audit log to revert file-system modifications resulting from the intrusion. To revert operations, the resolver *selectively* replays legitimate file-system operations on the tainted objects.

The rest of this section describes the design of the Taser system by first presenting the Taser recovery model. Then it describes the auditor, analyzer and resolver components of Taser. The overall system, as presented in this section, correctly reverts all file-system modifications resulting from an intrusion. Unfortunately, it also results in a large number of false dependencies, leading to legitimate operations being marked tainted. Section 3 presents enhancements that improve the accuracy of the recovery process.

name op	: name id	$\rightarrow$ directory name id, name
content op	: object id	$\rightarrow$ content
attribute op	: object id	$\rightarrow$ attribute

Table 1: The Recovery Model

2.1 Recovery Model

Taser assumes a POSIX-compliant Unix file-system consisting of regular files, directories, symbolic links and device nodes, each of which has three types of information associated with it: name, content, and attributes. Taser treats file name, content and attributes as separate objects during recovery, and assumes that operations on each object are independent. For example, it assumes that name operations occur independently of content or attribute operations. Separating file-system operations helps in optimizing name and attribute recovery as discussed later in Section 2.4, and it allows finer-grained analysis policies as discussed in Section 3.1.

Taser distinguishes between a file object and a name object because Unix files can have multiple names. It assumes that an object id uniquely identifies a file object and a name id uniquely identifies a name object. In Unix file systems, the object id contains the inode number of the file. A name id is associated with exactly one object id and this association is immutable over the lifetime of the system. In contrast, an object id can be associated with multiple name ids because a file object can have multiple names. Additional requirements on these identifiers, such as uniqueness over time, are described later in Section 4. File names in a Unix file system are stored as part of the contents of directories. Taser recovers these contents indirectly during name recovery.

Taser enforces file attributes such as permissions and ownership at the file object (or inode) level, immaterial of the name by which the file is accessed. For example, modifications to file permissions (e.g., via `chmod`) are assumed to occur directly on the inode rather than via the name of the file.

To formalize this model, we define the three types of file-system operations as the mappings shown in Table 1. A name operation (e.g., `rename`) creates, modifies or removes the mapping between a name object and the pair (directory name id, name). A directory name id is a name id associated with a file object of type directory. A content operation creates, modifies or removes the mapping between a file object and its content, and similarly for the attribute operation. These definitions make the three different types of operations on an object independent of other operations on the same or other objects provided that three *consistency requirements* imposed by the file system are met:

1. The name or object id must exist for a successful operation.
2. The directory name id must exist for a successful name operation.
3. The name mapping must be one-to-one, i.e. two different name objects in the same directory must map to different file names at any given time.

2.2 Auditor

The auditor tracks operations on three types of kernel objects: processes, files and socket connections. In particular, the auditor captures the names and *all* the arguments of system call operations related to process management, file system operations and networking in an audit log. We assume that the auditor captures the identity of the objects (e.g., process id, file object id, etc.) associated with

system calls and it audits concurrent system call operations correctly. We also assume that the auditor is not corrupted as a result of an attack. Section 4 describes the prototype implementation of the auditor and discusses the measures we take to ensure that these assumptions are held.

2.3 Analyzer

The analyzer determines the set of tainted file-system objects by creating dependencies between sockets, processes and files based on entries in the audit log. Socket connections form initiating points for remote attacks, processes issue operations that create other dependent processes or files, and file accesses cause additional dependencies, and, in addition, files are the persistent state of the system that need to be recovered. The following sections describe how dependencies are created and then present the tainting algorithm used by the analyzer.

2.3.1 Dependency Rules

We say that a dependency is caused when information flows from one kernel object to another via a system-call operation. A dependency is denoted by $O_s \xrightarrow{op} O_d$ where O_s is a source object, O_d is the dependent object, and *op* is the relevant operation. For example, when a process writes to a file, the file becomes dependent on the process. Similarly, a process becomes dependent on a file when it reads the file. Table 2 shows the dependency rules between the kernel objects that are considered by the analyzer. These rules are used to taint a dependent object when the source object is tainted. Each dependency, which always involves a process, is caused by the type of operations shown in the corresponding row. The last column of Table 2 shows some of the key system call operations that constitute each type of operation.

A process to process dependency occurs when a child process is forked, which captures a tainted process hierarchy, and when IPC and signal-based communication occurs between processes. A process to file dependency occurs when a process writes to the content, name (e.g., creating or removing a file name), or attributes (permissions, ownership) of a file. A file to process dependency occurs either when a process executes a file or reads the content, name or attributes of the file. For example, suppose a directory's attribute is tainted. A process accessing that directory will then become tainted. When a process reads or accesses a path name, a dependency occurs from every component of the path name to the process since each component is considered a separate object in the Taser recovery model. A process to socket dependency occurs when a process writes to a socket, and a socket to process dependency occurs when a process reads from a socket.

2.3.2 Tainting Algorithm

The tainting algorithm derives the set of tainted objects using the audit log, the dependency rules shown in Table 2 and an initial set of tainted objects, known as detection points, that are provided by an intrusion detection system (IDS) or an administrator. Detection points can either be the source of an attack (e.g., a malicious socket connection that originates an intrusion), or the result of an attack (e.g., some strange files identified by a host IDS).

When the detection points are not the source of an attack, the algorithm goes into an initial tracing phase that starts from the detection points and scans the audit log backwards to trace the source objects of the attack. The algorithm then switches to the propagation phase that starts from the source objects of the attack and scans the audit log forwards and taints objects affected by the intrusion. Below, the tracing and propagation phases are described in more detail.

Dependency Rule	Type of Operation	Operation
Process → Process	Fork	fork, vfork
	IPC, Signals	pipe, kill, mmap
Process → File	Write file content	creat, truncate, unlink, write
	Write file name	creat, link, symlink, rename, unlink
	Write file attributes	create, unlink, chown, chmod
File → Process	Execute	execve
	Read file content	read
	Read file name	open, truncate, chown, chmod
	Read file attributes	open, truncate, chown, chmod
Process → Socket	Write	write, socketcall, sendfile
Socket → Process	Read	read, socketcall

Table 2: Dependency rules between processes, files and sockets

Tracing Phase. To identify the source objects of an attack, this phase starts with a set of detection points and traverses dependencies in reverse causality order. The trace phase is given a conservative estimate of the attack start time by an administrator, and the reverse traversal starts from the last occurrence of the detection points in the audit log, and it proceeds until the start-time.

The output of this phase is a set of objects. This set helps an administrator choose objects that are deemed to be attack sources. This step, while requiring manual feedback, allows the administrator to limit the set of tainted source objects. For example, the administrator might know that a process but not an entire session is tainted. Grouping the output according to various criteria, such as processes that have accepted remote connections, setuid processes or files, may aid the administrator perform this analysis.

Propagation Phase. The propagation phase starts with a set of source objects determined by the tracing phase or provided by an IDS or an administrator. This phase traverses dependencies forward to compute a causal dependency graph that consists of the transitive closure of all tainted objects. The propagation phase starts from the attack start-time and uses the dependency rules to mark objects as tainted. When an operation shown in Table 2 occurs, and the source object is tainted, the taint status propagates to the dependent object. The dependent object is then marked tainted, and the tainting time is recorded. This process is repeated until the tainted status does not propagate to other objects any further. At the end of this phase the analyzer outputs the set of tainted file-system objects.

The propagation phase treats file content, name and attributes as separate objects. Section 3 describes how this separation helps implement optimistic analysis policies that reduce false dependencies.

2.4 Resolver

The goal of the resolver is to revert tainted file-system operations but preserve legitimate operations. It takes as input a file-system snapshot, the set of tainted file-system objects generated by the analyzer, and the audit log created by the auditor. To revert operations, the resolver uses a *selective redo* algorithm that only replays legitimate operations in the log that occur on the tainted objects. The resolver assumes that recovery starts with an immutable file system so that the file-system state does not change during recovery.

The resolver only considers successful legitimate operations that modify the file system; it ignores read-only operations or operations that returned with a failed status. It is possible that that these operations would have yielded different results (e.g., a failed legitimate operation could have succeeded) if the intrusion had not occurred. However, the resolver does not know the semantics of the processes that issued the legitimate operations, and hence does not attempt to predict process behavior if tainted operations had not occurred. Similarly, the resolver preserves the effects of all legitimate operations even though it is possible that a legitimate operation may have failed if the intrusion had not occurred (e.g., writes to a file made accessible by a tainted operation).

The rest of this section first considers a simple recovery algorithm based on redo logging. Then it presents selective redo, an optimized redo algorithm that is used by the resolver.

2.4.1 *Simple Redo Algorithm*

In the simple redo algorithm, recovery starts with a file-system snapshot and sequentially replays the file-system modification operations captured in the audit log. Only the legitimate operations should be replayed since the effects of the tainted operations should be ignored. The resolver uses the set of tainted file-system objects (generated by the analyzer) to differentiate between legitimate and tainted file-system modifications operations in the audit log. In particular, modification operations to a file-system object that occur after the time the object was tainted are marked as tainted operations and are not replayed. This simple redo solution is correct because the dependency rules in Table 2 ensure that legitimate operations do not depend on tainted operations. Unfortunately, replaying all legitimate operations can be a slow process.

2.4.2 *Selective Redo Algorithm*

The Taser selective redo algorithm makes two optimizations to improve the performance of the recovery process. First, we observe that the file-system state at recovery time has the correct state for all non-tainted objects. Therefore, the resolver starts the recovery process with the file system at the recovery time instead of the file system at the snapshot time, and it *selectively* replays legitimate operations *only* on tainted objects. To recover a tainted object, the resolver obtains an initial version of the object from the file-system snapshot and sequentially replays the object's legitimate modification operations since the snapshot.

A second optimization takes advantage of the Taser recovery model and performs recovery for file name, content and attribute operations *separately*. Separating file-system operations helps in optimizing name and attribute recovery. At each name or attribute operation, the auditor captures the complete state of the object as shown in Table 1. For example, it captures all the attributes (permission, ownership) of a file after an attribute operation. As a result, a sequence of attribute and name operations can simply be replaced by the last operation during recovery. Therefore, the resolver recovers a tainted attribute or name by replaying the *last legitimate* operation on that attribute or name. In contrast, to recover file contents, the resolver replays all legitimate content operations starting from the snapshot until the first tainted operation. It does so because, for storage efficiency, the audit log does not store the complete state of the content mapping at each operation. Note that name recovery implicitly recovers directory contents.

The resolver performs name recovery before content or attribute recovery. This ordering helps meet the consistency requirements discussed in Section 2.1. Intuitively, name recovery sets up a virtual, consistent name space for the recovered file system, and then content and attribute operations are performed on this name space.

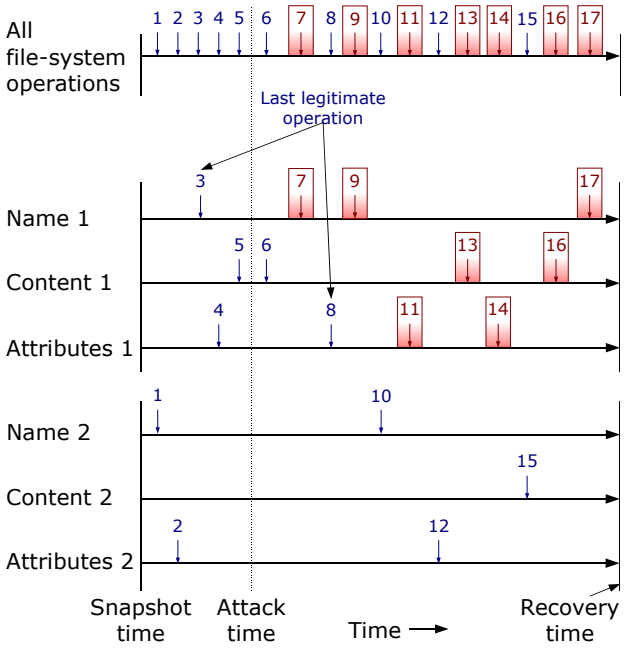


Figure 1: Separating content, name and attribute operations

Figure 1 presents an example to illustrate the selective redo recovery algorithm. This figure shows the snapshot time when the file-system snapshot is taken, the attack time when an attack occurs, and the recovery time when the attack is detected and intrusion recovery is started. All the file-system operations are shown at the top of the figure. This example shows how the file-system operations can be separated into name, content and attribute operations for two files, File 1 and File 2. Operations that occur after the attack time are marked tainted and are shown in boxes. Note that the dependency rules in Table 2 ensure that after the first tainted operation, all operations on a tainted name, content or attribute object are marked tainted.

Recovery starts with the file-system state at the recovery time. Note that File 2 is untainted and no operations need to be redone for this file. In contrast, File 1 has to be recovered. Name 1 can be recovered in a single step by replaying operation 3. Similarly Attributes 1 can be recovered by replaying operation 8. Finally, Content 1 is recovered by replaying operations 5 and 6. In this example, selective redo requires replaying four legitimate operations, whereas the simple redo algorithm requires replaying all ten legitimate operations. In general, selective redo is beneficial if the footprint of the attack is small compared to the total number of legitimate modification operations since the snapshot.

3. DESIGN ENHANCEMENTS

The Taser system presented in the previous section reverts all tainted objects to their legitimate states, but it can also result in a large number of false dependencies leading to legitimate objects being marked tainted. These objects are then unnecessarily reverted to a previous state. While the dependency rules presented in Table 2 cause information flow, it is unclear, without detailed program analysis [21], whether the dependencies that are created are real dependencies. For example, a signal sent from a tainted process to another process may occur as part of normal activity, and hence the dependent process should not be marked tainted. Simi-

Policy	Description	Conflicts
Conservative	All operations shown in Table 2	none
NoI	Ignores IPC, signals	none
NoIA	Ignore reading file attributes	attribute conflicts
NoIAN	Ignore reading file attributes and names	attribute, name conflicts
NoIANC	Ignore reading file attributes, names, content	attribute, name, content conflicts

Table 3: Optimistic analysis policies and classes of conflicts

larly, applications may read and write from `/dev/null` but this file does not cause any explicit information flow. This section describes enhancements to the analyzer and the resolver that improve the accuracy of Taser.

3.1 Analyzer Enhancements

The analyzer enhancements reduce the possibility of tainting legitimate objects by relaxing the application of dependency rules. The analyzer relaxes dependency rules in three different ways, optimistic analysis policies, dependency intervals and white lists, that are described below. Relaxing the rules can lead to conflicting recovery actions. The resolver enhancements discussed in Section 3.2 handle such conflicts. Furthermore, the analyzer enhancements can miss tainting an attacker's operations. This issue is discussed further in Section 5.

3.1.1 Optimistic Analysis Policies

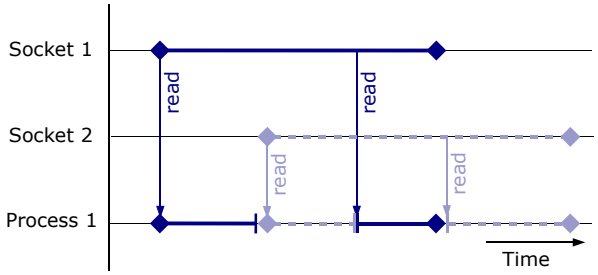
The analyzer described in Section 2.3 uses a conservative taint analysis policy that takes into account all dependency rules listed in Table 2. This policy correctly identifies all tainted kernel objects, but it generates a large number of false dependencies that cause legitimate objects to be marked tainted. To reduce this problem, and in turn enable the recovery system to preserve more legitimate operations, we extend the analyzer with optimistic taint analysis policies that ignore some of the dependency rules listed in Table 2.

Table 3 shows the conservative analysis policy implemented by the original analyzer and four optimistic analysis policies. The names of the optimistic policies reflect the set of dependency rules that the optimistic policy ignores. For example, the NoI policy ignores IPC and signal operations. Similarly, NoIA, NoIAN and NoIANC successively ignore reading file attributes, file names, and file contents. The NoIANC policy is the most optimistic policy. It includes a minimal set of operations that we consider essential for creating tainting dependencies: fork that creates the tainted process hierarchy, file or socket writes by a tainted process, execution of a tainted file, and reads from a tainted socket. Table 3 also shows the conflicts that may be generated by the different optimistic policies. These conflicts and their resolution are discussed in Section 3.2.

Although the burden of choosing an appropriate policy for an intrusion lies with the system administrator, in practice, we find that running all the policies and comparing their results helps in quickly determining appropriate recovery actions. This issue is discussed further in Section 5.

3.1.2 Dependency Intervals

Until now, the analysis policies were based on dependencies between *different* objects, which we call *inter-object* dependencies. Furthermore, similar to previous dependency analysis approaches such as the Backtracker [12], we have implicitly assumed that an



Objects are shown on the left side of the figure. Inter-object dependencies are shown as vertical arrows while intra-object dependencies are shown as horizontal lines. The start and end of an intra-object dependency interval is shown with a ♦ symbol. Dependency intervals can be multiplexed over time as shown by the | symbol for Process 1. For this process, reading from different sockets starts the different intervals.

Figure 2: Inter- and intra-object dependencies

intra-object dependency exists within an object for the life-time of the object. With this conservative approach, once an object is tainted, it remains tainted forever. For example, a process remains tainted until it dies. This approach avoids missing dependent tainted operations but, unfortunately, it can be too coarse-grained, especially when objects exist for long periods. For example, a server process often performs activities that are logically distinct, one for each connection. Tainting based on this long running process will generate false dependencies between these unrelated activities simply because the same process issues them.

To reduce these false dependencies, we enhance the analyzer to optimistically limit intra-object dependencies to certain intervals so that each interval is considered independent of the other intervals. Below, we explain how these intervals are defined for each of the kernel objects.

Processes. For processes, we consider two cases: 1) *source* processes that communicate using a socket connection that is initiated by a remote source, and 2) all other processes. For the latter case, we define the dependency interval as simply the process lifetime. For the former case, a source process can be the initiating point for external attacks. Such a process or its parent is typically a demultiplexing point for large numbers of unrelated activities (e.g., a server process). Hence, creating different intervals for the unrelated activities can significantly avoid false dependency sharing.

To derive intervals for a source process, the analyzer examines successful socket reads where a process reads remote data. A read from a different remote socket indicates that the process switches "context" to work for another unrelated activity. Such a read terminates the current interval and starts another interval. This interval method can be used for various common server models such as separate processes started by Inetd, worker processes that handle different multiple connections, and event-driven servers that multiplex the activities of different connections.

Figure 2 illustrates inter- and intra-object dependencies. Similar to the notation used in Magpie [2], the start and end of an intra-object dependency interval is shown with a ♦ symbol. However, unlike Magpie, the analyzer uses directed dependencies. The figure shows two different intervals for Process 1 based on whether the process is serving Socket 1 or Socket 2. Each of these intervals is multiplexed over time as shown by the | symbol.

File name. With the default analysis policies, once a name in a directory is tainted, it remains tainted forever. For example, say a tainted name is removed and a legitimate process creates the same name again. Then this name will still be tainted. With dependency intervals, a file name interval starts when a new file name is created and it ends when the file name is removed. A new file name is created either when a file is created or the `link` system call is used to create a new name for an existing file. A `rename` operation is treated as a name removal followed by the creation of another name.

Files content. For file content, the dependency interval starts when a new file is created and it ends when the file is removed. In addition, the interval ends and starts another interval if the data of an object is completely overwritten. For example, the complete truncation of a file starts a new interval since the truncation stops any file content related dependency.

File attributes. For file attributes, the dependency interval starts when a new file is created and it ends when the file is removed.

Sockets. For sockets, the dependency interval is simply the lifetime of the socket. A new connection on the same port creates a different object.

The analysis policies shown in Table 3 can optionally use the dependency intervals described above. With these intervals, the tainted status of an object ends with the end of the interval. Hence, if this object interacts with a dependent object after the interval, the taint status is not propagated to the dependent object. Such an interval policy is more optimistic than the corresponding default policy because it limits the time over which an object can create causal dependencies.

3.1.3 White lists

In practice, we find that some analysis policies that create dependencies based on reading tainted files can cause widespread tainting. For example, many processes read and write shared files such as `/var/log/wtmp` and `/dev/null`. To avoid creating such false dependencies, Taser provides a white list mechanism that allows an administrator to designate a list of files that are ignored by these policies. A white-listed file that is written by a tainted process becomes tainted and will be reverted, but this object does not propagate its tainted status to other objects.

3.2 Resolver Enhancements

While the analyzer enhancements described in Section 3.1 help reduce false dependencies, they can, however, cause *conflicts* when legitimate operations depend on tainted operations. In this case, reverting the tainted operations may result in reverting legitimate file-system operations or the loss of legitimate file-system objects. For example, with the NoIAN policy, which ignores tainted file attributes and names, a conflict occurs when a legitimate file is created inside a tainted directory. A recovery action that simply removes the tainted directory will lose the legitimate file creation. We consider such a recovery action as having failed because our goal is to preserve all legitimate operations.

Conflicts occur when an operation reads a tainted file-system object, and this read is ignored by some analysis policy shown in Table 3. As a result, legitimate operations can occur after tainted operations as shown in Figure 3. Table 3 shows the classes of conflicts that can arise with each analysis policy. Conflicts arise only when file-system dependencies are ignored. For example, ignoring attributes causes attribute conflicts, ignoring names cause name

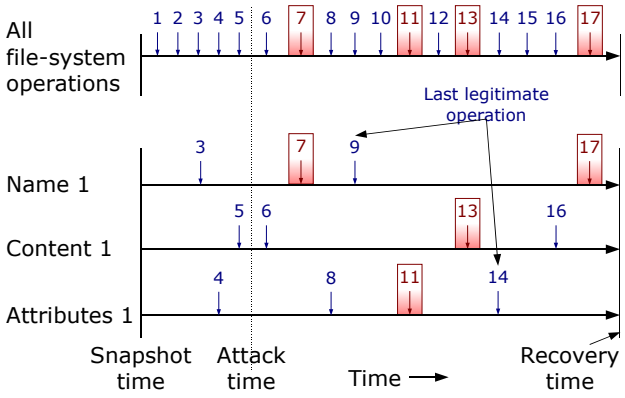


Figure 3: Legitimate operations occur after tainted operations

conflicts, etc. The more optimistic policies, by ignoring more dependencies, can cause more conflicts. In the example above, reading a tainted directory does not taint the process or the file creation. This conflict would occur with the NoIAN and the NoIANC policies that ignore reading file (or directory) names.

Table 4 provides a finer classification of conflicts, which allows designing resolution policies suited for each type of conflict. The file-system operations are divided into elementary operations and the types of conflicts are based on these operations. A conflict arises when a legitimate operation, shown along the top row, reads an object that was modified by a tainted operation shown along the left column. Directory-related operations are not shown in the table but are discussed below.

Next, we describe the different types of name, content and attribute conflicts, and the resolution policies implemented by the resolver. The resolution policies help *isolate* conflicting tainted operations because these operations cannot be completely reverted.

3.2.1 Name Conflicts

Name-create conflict. A name-create conflict occurs when a legitimate name creation operation accesses a tainted name. For example, an administrator renames a file created by an attacker. Recall that a tainted name is recovered by simply replaying the last legitimate operation on that name. This operation may conflict with a previous tainted name operation. For example, in Figure 3, operation 9 may generate a name that depends on the name produced by tainted operation 7. This name conflict occurs with the NoIAN and NoIANC policies that ignore dependencies caused by reading tainted file names (shown in Table 3). The resolver also ignores this conflict because the conflict does not violate any of the consistency requirements described in Section 2.1.

Name-remove conflict. A name-remove conflict occurs when a legitimate operation removes a tainted name or a directory containing a tainted name. These conflicts could be ignored because the object was legitimately removed previously. However, it is possible that a user would not have removed this object if the name had not been tainted. For example, the user may have removed a legitimate file that was renamed to an unusual name by a tainted operation. Hence, for these conflicts, the object is recreated with the name that has a `.removed` extension so that it can be inspected manually.

Tainted Operations	Legitimate Operations			
	Name Create	**Name, Object Remove**	**Content Update**	**Attribute Update**
Name Create	name-create conflict	name-remove conflict	name-access conflict	name-access conflict
Name, Object Remove	name-recreate conflict	not possible	not possible	not possible
Content Update	non-conflicting	content-remove conflict	content-access conflict	non-conflicting
Attribute Update	attribute-access conflict	attribute-remove conflict	attribute-access conflict	attribute-access conflict

Table 4: Types of conflicts caused by different legitimate and tainted operations

Name-access conflict. A name-access conflict occurs when a legitimate operation updates the content or attributes of a file with a tainted name, or modifies a file under a tainted directory. In this case, the file or the directory has seen no legitimate name operations. This object should be removed, but the relevant legitimate operations should be recovered. For example, an administrator may have created a legitimate file under a tainted directory. Simply removing the tainted name would violate one of the first two consistency requirements described in Section 2.1. To resolve this conflict, the tainted name of the object is isolated, instead of being reverted, and recovered with a `.nonexistent` extension. This extension indicates that the name was not created legitimately. At the end of recovery, a list of these suspect objects is provided so that the user can inspect the objects and take appropriate actions.

Name-recreate conflict. A name-recreate conflict occurs when a legitimate operation recreates a name that was removed by a tainted operation. For example, the administrator may recreate a legitimate file that was removed by an attacker. Simply recreating the removed tainted object leads to two different but legitimate objects with the same name, which violates the third consistency requirement. If the recovery action recreates the same object such as via multiple names of a file, then the conflict is ignored. Otherwise, the resolver recreates the previous objects with the same name but with a version number extension.

3.2.2 Content Conflicts

Content-access conflict. A content-access conflict occurs when a legitimate operation updates the tainted contents of an object. Recall that the resolver replays the legitimate content operations starting from the snapshot until the first tainted operation. For example, operations 5 and 6 would be replayed to recover Content 1 in Figure 3. Any legitimate operation after the first tainted operation causes a content-access conflict because we assume that content operations always read contents before modifying them. Content-access conflicts need to be fixed manually since file contents are typically unstructured. An alternative is to use application-specific conflicts resolvers [19, 13, 29].

Content-remove conflict. This conflict occurs when a legitimate operation removes an object whose content is tainted. Similar to the reasons for storing objects that are involved in a name-remove conflict, objects involved in a content-remove conflict are also recreated with a name that has a `.removed` extension so that the contents of the object can be inspected manually.

3.2.3 Attribute Conflicts

Attribute-access conflict. An attribute-access conflict occurs when a legitimate operation (other than remove) accesses the tainted attributes (permission or ownership) of an object. Recall that a tainted attribute is recovered by simply replaying the last legitimate operation on that attribute. For example, Figure 3 shows the last legitimate and the last tainted operations (operations 14 and 11) on Attributes 1. In this case, since the last operation is legitimate, nothing needs to be done for recovery, otherwise, operation 14 would be replayed if there were tainted operations after it. Similar to name-create conflicts, the resolver ignores this conflict because it does not violate the consistency requirements.

Attribute-remove conflict. This conflict occurs when a legitimate operation removes an object whose attributes are tainted. This conflict is resolved in a similar way to name-remove conflicts by recreating the object with a name and that has a `.removed` extension and with legitimate attributes.

3.2.4 Global Conflict Resolution

Recall from Section 2.4 that the resolver performs name, content and attribute recovery separately. Typically, conflict resolution is performed as part of the corresponding recovery operation. For example, name conflicts are resolved as part of name recovery, etc. However, certain conflicts must be resolved globally after all recovery actions have been generated. For example, a name-access conflict occurs when a legitimate operation updates the content or attributes of a file with a tainted name. This name conflict can be detected and resolved only after the name, content and attribute recovery algorithms have been executed. In particular, the tainted name cannot be removed during name recovery if the object has legitimate content or attribute modifications.

4. IMPLEMENTATION

The Taser recovery system consists of the auditor, analyzer and the resolver. Each component is discussed below.

4.1 Auditor

The auditor tracks the operations of three types of kernel objects: processes, files and socket connections. The prototype implementation uses the Forensix system [8] to audit all kernel operations related to process management, file system and networking. Figure 4 shows the Forensix architecture. The target system, which provides services to the public network, is potentially vulnerable.

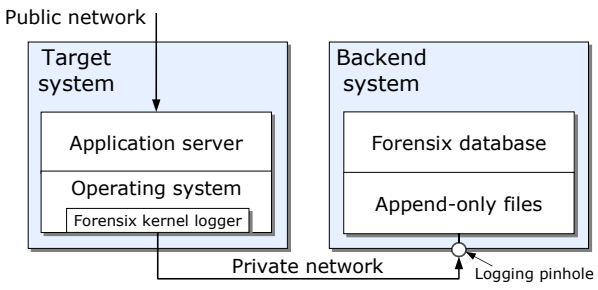


Figure 4: The Forensix architecture

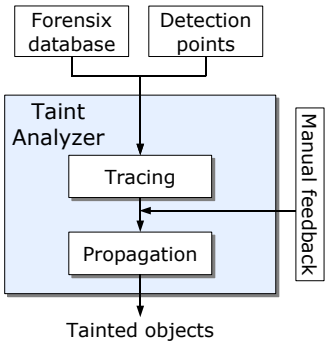


Figure 5: The Analyzer architecture

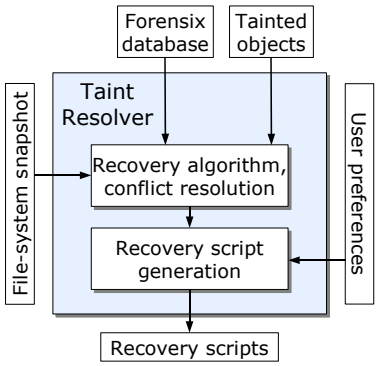


Figure 6: The Resolver architecture

The Forensix kernel logger is implemented as a module that loads into a Linux kernel. It logs all process management, file system and networking system calls and uses the Linux security modules facility [31] to capture additional information that helps disambiguate the identity of the kernel objects accessed during system calls. This approach provides accurate ordering of operations and avoids race conditions during auditing [6].

The logger transmits the target's system-call audit log over a dedicated network to a secured backend system where the log is stored in append-only files. Separating the backend from the target machine helps ensure that the logged information cannot be destroyed easily. In the backend system, the audit log is batch-loaded into a database periodically or on demand. This log is then used by the analyzer and the resolver that operate entirely on the backend.

We assume that the logging system on the target is not corrupted as a result of an attack. Since Forensix runs in the kernel on the target system, this implies the assumption that the applications but not the kernel on the target are vulnerable. We examined statistics of known vulnerabilities for three distributions of Redhat Linux (Fedora Core 1, 2 and 3) and found that the breakdown of the number of kernel to application-level vulnerabilities in these three systems was 9/74, 9/132 and 6/112 [24], i.e. kernel vulnerabilities account for 10% or less of all vulnerabilities. While security statistics are never conclusive [16], the numbers above substantiate our belief that it is easier to secure a single piece of kernel code than all applications combined. To reduce the risk of kernel intrusions, Forensix uses LIDS [32] to disable 1) user-level writes to kernel memory, 2) user-level writes via the raw disk interface, 3) writing to the kernel or Forensix binary files, and 4) the loading of kernel modules. These simple measures make current kernel-level rootkits ineffective [7, 23].

To facilitate analysis and recovery, each kernel object of interest (sockets, processes and files) must be assigned an identifier that remains unique over time. For sockets and processes, the Forensix kernel logger attaches a creation time-stamp to the socket and the process id. To track operations on a file object, the logger uses an object identifier, which for Unix-based files is the inode number. However, since inode numbers can be reused after an object is removed and these numbers are not unique across devices, the logger uses the three tuple (device, inode, generation number) to uniquely identify file-system objects. We call this tuple the object id. The generation number is available in most current Unix file systems and serves the same purpose as a time stamp.

The object id, by itself, is not sufficient for tracking the names of an object since file objects can have multiple names. To track names, the backend maintains a name id for each name of the file that consists of the tuple (object id, creation id). When a file object is created, it is assigned a starting creation id and when a new name for the file is created (e.g., with the `link` system call), this new name is given a new creation id. The creation id does not change when a name is updated, e.g, with the `rename` system call. The removal of a name ends the lifetime of the name id associated with that name. The name id approach allows tracking name operations independently of file object or inode (content, attribute) operations.

4.2 Analyzer

Figure 5 shows the architecture of the analyzer. The implementation of the analyzer is relatively straightforward. Starting from the detection points, all the relevant system call operations in the audit log, available from the Forensix database, are parsed to create the dependency information. This parsing is performed in reverse time order during tracing and forward time order during the propagation phases. As an optimization, dependencies between legitimate objects are not created since they are not needed. The manual feedback step is needed between the tracing and the propagation phases to choose attack sources. Although the Forensix database contains IPC and signal operations, the analyzer currently ignores these operations while creating dependencies. As a result, it implements all policies in Table 3 except the Conservative policy.

4.3 Resolver

The resolver obtains the tainted objects from the analyzer and then uses the Forensix database to derive the legitimate file-system operations on these objects. Figure 6 shows the architecture of the resolver. The recovery algorithm implements and uses several mapping tables in the Forensix database. These tables store an archive of the mapping information described in Table 1, and they allow creating snapshots of the file system, including the state of the file system just before the attack time or at the recovery time on the backend machine.

The output of the recovery and the conflict resolution algorithms is a set of recovery actions that take the immutable file system at recovery time to a recovered state that reverts the effects of tainted operations. A final recovery script generation phase orders the recovery actions so that they can be executed consistently. For example, suppose that name A at recovery time must be renamed to name B. However, name B exists at recovery time and must be renamed to name B.new. Then the script generation phase orders the second action before the first one. This phase takes into account user preferences such as whether old object versions should be kept or certain files can be ignored during recovery (e.g., editor backup files). The recovery script is executed on the target machine.

5. EVALUATION

Our evaluation of Taser consists of two parts. First, we evaluate the accuracy of the system to determine whether Taser can effectively recover from a wide range of intrusion and management errors. Second, we evaluate the performance of the system by measuring the time to perform analysis and recovery and the performance and space overheads of auditing. Below, we first describe the experimental method and setup and then present our evaluation.

5.1 Experimental Method

We evaluated Taser using several scenarios that are described in Section 5.3. For each scenario, the trace phase of the analyzer uses the most conservative taint analysis policy that we have implemented (the NoI policy) so that the attack source is not missed. The propagation phase uses three different policies: NoI (ignores IPC and signals), NoIAN (ignores IPC, signals, file name and attributes) and NoIANC (the most optimistic policy that ignores all previous dependencies and file content). We don't present the results for the NoIA policy because preliminary analysis showed that these results are similar to the NoI policy.

To evaluate accuracy, we use two metrics: 1) the number of false positives, which are legitimate operations that are marked tainted and reverted to a previous state, and 2) the number of false negatives, which are the attacker's operations that are not caught and for which no recovery action is taken. For each metric, we count the number of distinct name and attribute recovery actions, and the number of distinct file objects that require content recovery. To calculate the number of false positives and negatives, we need to determine the *correct* number of recovery actions. To this end, we manually examine the name, contents and attributes of every file that has changed since the attack. This information is available from the Forensix database.

To evaluate the analysis policies, we also implemented a base case Snapshot policy in the analyzer. This policy is implemented by tainting all processes and their file operations after the attack time. This policy can have false positives only, which is the number of legitimate actions after the attack that are reverted. The Snapshot policy represents the best possible case for a traditional snapshot-based file-system recovery approach since it assumes that a snapshot is taken just before the attack.

The NoI and the NoIAN policies create dependencies due to reading file content. In general, they can generate large numbers of false positives due to false dependencies caused by reading character device files and log files. As a result, we ran experiments for these policies using a small set of white list files that are determined experimentally. Unless explicitly mentioned, the default white list for these policies consisted of the following: 1) all character device files such as `/dev/null` and `/dev/pts/0`, 2) the log files `/var/log/wtmp*`, `/var/run/utmp`, `/var/log/lastlog`, and 3) shell history files. There is no reason to use a white list with the Snapshot policy, and the white list does not make any difference for the NoIANC policy because this policy does not taint objects via reading file content.

By default, we used intra-object dependency intervals defined in Section 3.1.2 for all the policies. One of the scenarios used a server process, and for this scenario, we show that using a single dependency interval causes a large number of false positives.

5.2 Experimental Setup

The experimental setup consists of a target and a backend machine. The target machine is an AMD Athlon 2600+ machine with 512 MB memory. It runs stock Redhat 7.2 together with the Forensix logger. It contains four vulnerable services or executables: the samba and the wu-ftpd daemon that allow remote root exploits, and the sendmail and the pwck-setuid programs that allow local root escalation exploits. The backend machine is an Intel Pentium 4 2.4 GHz CPU with Hyper-Threading and 512 MB memory. It runs Redhat Fedora Core 3 and uses the MySQL version 4.1.10 database for storing the audit data.

We evaluated Taser for approximately a week. During this time, we ran a popular web-based photo album application called Gallery on the target system. To induce concurrent activity and to load the system, we simulated users interacting with Gallery with a client-side Galhogger program that was run continuously for the entire week. Galhogger simulated an anonymous user that browses the albums every 2 seconds on average, and 5 registered users that each modify the albums every 3000 seconds on average.

5.3 Recovery Accuracy

We use six different scenarios to evaluate the accuracy of Taser. For each scenario, we performed recovery one-day and one-week after the incidents to evaluate the accuracy of Taser over time. Below, we describe each scenario, the correct recovery actions for each scenario, the initial detection points and the results of using Taser. The accuracy results are summarized in Figure 5. This figure shows the false positives and negatives (separated by a comma) for the different taint analysis policies under each of the scenarios. We performed recovery for all the scenarios after all the attacks have been performed. As a result, the false positive numbers for the Snapshot policy are similar for the different scenarios.

5.3.1 Illegal storage

Scenario: A user logs into the system and launches the pwck local escalation exploit. This exploit allows the user to get a root shell. This attacker creates a new root account root100 by directly writing to the `/etc/passwd` and the `/etc/shadow` files. This attacker creates a directory under another user's directory (as root) and downloads 500 illegal pictures into this directory. Finally, he downloads a binary `ls` program from the user-level Ambient's Rootkit (ARK) in the user's bin directory to hide the existence of the illegal directory. Later, the victim user logs in, uses the trojaned `ls` program and creates two files in his home directory. The attacker logs back in as root100 after two days and downloads two more pictures into the hidden directory.
Correct recovery actions: Remove all the illegal pictures and the hidden directory, the trojaned `ls` binary, and the home directory of the attacker's root100 account. In addition, the legitimate versions of the `/etc/passwd` and `/etc/shadow` files need to be recovered.
Detection point: The trojaned `ls` program is detected by the victim and given to the trace phase.
Results: The trace phase detected the remote connection of the attacker and propagation is started using the remote host address. The NoI and the NoIAN policies with the default white list generated several false positives because `/etc/passwd` and `/etc/shadow` are written by the attacker and their content is read by all the following ssh login processes which also get tainted. As a result, all these activities would be reverted. We added `/etc/passwd` and `/etc/shadow` to the white list and re-ran these policies. Table 5 shows that two false positives occur with the NoI policy when recovery is performed after a day. These errors occur because the victim's shell process got tainted when it accessed the name of the trojan `ls` program. As a result, the two new files created by the attacker are tainted and will be removed by this policy. The other two policies have no errors when recovery is performed after one day.

Scenario	Recovery Actions	Recovery one day after attack				Recovery one week after attack			
		Snapshot	NoI	NoIAN	NoIANC	Snapshot	NoI	NoIAN	NoIANC
Illegal storage	507	633, 0	2, 0	0, 0	0, 0	4154, 0	7, 0	0, 2	0, 2
Content destruction	739	1877, 0	0, 0	0, 0	0, 0	5338, 0	0, 0	0, 0	0, 1
Unhappy student	167	1106, 0	2, 0	0, 0	0, 1	4617, 0	4, 0	0, 0	0, 1
Compromised database	3	814, 0	0, 0	0, 0	0, 2	2557, 0	0, 0	0, 0	0, 2
Software installation	350	1542, 0	1, 0	0, 0	0, 0	5006, 0	1, 0	0, 0	0, 0
Inexperienced admin	39	1366, 0	11, 0	11, 0	0, 0	4982, 0	11, 0	11, 0	0, 0
Inexperienced admin (single interval)	39	1366, 0	415, 0	415, 0	125, 0	4982, 0	701, 0	701, 0	126, 0

For each scenario, the second column shows the correct number of recovery actions as determined manually. These numbers are roughly the same one day or one week after the attack. The rest of the columns show the accuracy of four analysis policies in terms of false positives (a legitimate operation is marked tainted and reverted to a previous state) and false negatives (a tainted operation is not caught and no recovery action is taken for it). The accuracy numbers are shown separated by commas. The accuracy of the policies are shown one day and one week after each attack.

Table 5: Recovery accuracy

When recovery is performed one week after the attack, the NoIAN and the NoIANC policies have two false negatives (see Figure 5) because the attacker's second login (that occurred after more than a day) and the creation of the two new picture files is missed by these policies (they would have been caught if the password files were not in the white list). The NoI policy has no false negatives because the second login by the attacker accesses the tainted root100 directory which taints this session. However, it has additional false positives because the victim user performed other legitimate actions later in the week. Table 5 shows that the recovery results after a week are similar to the recovery results after a day, and therefore, we limit the discussion below to recovery results after a day for the rest of the scenarios.

5.3.2 Content destruction

Scenario: A software developer has been working on the files `src/project.c`, `hfiles/p1.h` and `hfiles/p2.h`. He has also saved a backup of the project.c file in `backup/project.c~.bak`. Another developer on the system launches the sendmail local escalation exploit to get the root shell. This attacker deletes the `project.c` and `p2.h` files. The victim notices that the `project.c` file is missing. He copies the backup file to the `src` directory and also moves the `p1.h` file to the `src` directory. Then, he deletes the `hfiles` directory and notifies the administrator.
Correct recovery actions: Remove numerous files generated by the sendmail attack, restore the deleted `p2.h` file in the `hfiles` directory, recover the original `project.c` file and deal with different versions of this file.
Detection points: The missing `p2.h` and `project.c` files.
Results: The trace phase detected the attacker's login process, which is used for propagation. The results show that none of the policies, except the snapshot policy, have any errors. The `hfiles` directory is recovered with a `.removed` extension since it was legitimately removed by the victim (a name-remove conflict). The original `project.c` file is recovered with a version number extension (a name-recreate conflict).

5.3.3 Unhappy student

Scenario: An attacker launches a remote attack on the wu-ftpd daemon running on the system and modifies the permissions of a grades file in a professor's home directory to be globally writable. Later, student A (an accomplice) with a regular account modifies the grades file in the professor's directory and also copies the professor's whole home directory into his own directory. Then, student B (another accomplice) logs in and copies the modified grades file into his home directory and creates two other files.
Correct recovery actions: Recover the original grades file in the professor's directory, restore the attributes of this file, and remove all copied files in both student A's and student B's home directories.
Detection point: The grades file that the professor finds is writable by others.
Results: The trace phase detected the remote attacker's root shell as well as student A's login session but not student B's login because B did not modify the grades file. The NoI propagation policy detected and tainted student B's shell process and the two files created by him which are false positives. The NoIANC policy on the other hand did not taint student B's operations, which leads to a false negative because the illegal copy of the grades file in student B's home directory is not removed. The NoIAN policy had no errors because this policy tainted B's copy operation but not B's entire shell process.

5.3.4 Compromised database

Scenario: Authenticated MySQL clients update a MySQL database running on a remote server. An attacker launches a remote attack on the Samba daemon running on the target system, gets a root shell and creates an SSH backdoor by writing his public key to root's `authorized_keys2` file. Later, other remote legitimate clients insert transactions into the database. After six hours, the attacker uses the ssh backdoor to log back into the machine. He issues a local MySQL query to remove some transactions from the database. After that, more legitimate clients update the database.
Correct recovery actions: Remove the attacker's ssh backdoor by removing his public key from the `authorized_keys2` file. In addition, recover two files associated with a MySQL table in the compromised database.
Detection point: Use the Snort [20] network IDS to detect the Samba attack. Since Snort does not give us enough host related information about the attack, we then use Forensix tools [8] to determine the root shell created by the attack.
Results: The trace phase started from the root shell and detected the attacker's first connection, which is used for propagation. The results show that none of the policies have any false positives. All of them recover the `authorized_keys2` file. The NoI and the

NoIAN policies detected the attacker's second login via the dependency caused by the tainted `authorized_keys2` file and then recovered the database by restoring the modified table files to the state right before the attacker's second login when he modified the database. The rest of the legitimate database writes were marked with a content-access conflict but we were unable to recover them. The Snapshot policy would revert the database to the state before the attacker's first login and miss *all* database writes. TasTaserer uses file-based recovery. This approach works for MySQL because MySQL is typically used in a non-transactional mode. For transactional databases, the database recovery logs would need to be incorporated in the recovery process [14, 17]. The NoIANC policy missed the attacker's second login and hence did not revert the attacker's delete transactions causing two false negatives.

5.3.5 Software installation

Scenario: Unlike the previous scenarios, the next two scenarios present and analyze system administration errors. Using a root account, we install RealPlayer 8 in the wrong directory which causes it to create many files and directories in this directory. In addition, it creates or updates various Netscape, KDE and Gnome configuration files or directories in `/root` including a `.netscape/-plugins` directory. Later, the root user browses the web with the netscape browser and downloads and saves a PDF reader plugin for Netscape in the plugins directory.
Correct recovery actions: All the RealPlayer files and directories should be removed and the configuration files should be restored.
Detection point: One of the RealPlayer files.
Results: The trace phase detected the process that ran the RealPlayer installer program, which is used for propagation. For this scenario, none of the policies generated any false negatives. However, the NoI policy has one false positive because the plugins directory is tainted and this policy taints and removes the PDF reader plugin. The NoIAN and the NoIANC policies do not taint the plugin but the plugins directory is recovered with a `.nonexistent` extension because this directory was created by the tainted process and it contains the legitimate plugin file. The user needs to retrieve this plugin file separately.

5.3.6 Inexperienced administrator

Scenario: The administrator uses the photo Gallery software (which is also used as background load in the experiments) to store his digital pictures and also creates an account for a guest user. The new account is set up with a weak password because the administrator expects the guest to change the password soon. Then, the administrator adds new albums and pictures under his account and logs off. Before the guest can change his password, an attacker at a remote site logs into the guest's account by using a dictionary attack. The attacker creates two new albums and uploads 14 pictures to the site and then views the administrator's albums. Later, the administrator views the albums and discovers inappropriate images in the two new albums. He contacts the guest user and finds out that the guest user did not create these albums.
Correct recovery actions: Remove the attacker's album and all related data (such as thumbnails) generated by Gallery.
Detection point: A directory containing an attacker's album.
Results: The trace phase detected the attacker's remote connection and propagation is started using the remote host address. Gallery maintains album, image, thumbnail and photo visit counters in a file hierarchy. Table 5 shows that there are 39 necessary recovery actions: 2 of which are to remove the attacker's albums, 14 are to remove the images, 16 are to remove the automatically generated thumbnails, 6 are to remove the data files generated by Gallery, and the remaining one is to recover a common data file called `albumdb.dat` that contains some global Gallery information. The NoI and the NoIAN policies generated many false positives because these policies create content-related dependencies and Gallery always reads the `albumdb.dat` file which taints all connections and their subsequent operations. We added the `albumdb.dat` file and a per-directory `album.dat` file to the white list and re-ran these policies, which then generated 11 false positives. In contrast, the NoIANC policy generated no false positives because the tainted status does not propagate via the `albumdb.dat` file. This scenario created several name-recreate and name-access conflicts due to the visit counts that are stored as files in albums. These do not affect application behavior after recovery except that some visit counts become stale.

Gallery runs on the Apache server that uses a worker model for servicing requests. The experiment above was performed using multiple dependency intervals, one per connection, for the Apache worker processes. We perform the same analysis but this time create a single interval for the worker processes. The last row of Table 5 shows that the single interval policy results in false positives even with the NoIANC policy, the most optimistic policy, because once a worker process is tainted by the malicious connection then it remains tainted even if it later services a legitimate connection. We find that Apache kills and re-spawns a new worker thread regularly. Otherwise, with the single interval policy, all Gallery actions after the attack would be reverted.

5.3.7 Kernel rootkits

Recall that Forensix avoid compromises to the kernel logger by securing the target operating system with LIDS [32]. To confirm that this approach is successful in thwarting attacks, we ran two kernel-level Linux rootkits, Knark [15] and SucKIT [23]. Knark loads a kernel module into Linux, while SucKIT directly issues user-level writes to kernel memory. Both these rootkits run unsuccessfully on the target system. They do issue some file-system modification operations before failure such as downloading, uncompressing and compiling the attack code as well as installing some trojan binaries such as `/sbin/init`. These operations are correctly recovered by Taser and we do not show any further results for these attacks.

5.3.8 Discussion of accuracy results

The previous sections have evaluated the accuracy of Taser in terms of correctly recovering legitimate data under varied scenarios. In this section, we highlight several key results shown in Table 5. First, the table shows that the analysis policies typically achieve high accuracy. There are few false positives or negatives even though many tainted and legitimate operations occur in the system as seen in the recovery actions and the snapshot columns respectively.

Second, the accuracy results do not vary significantly when recovery is performed one day or one week after the attack. This result shows that attacks do not have to be detected immediately for useful intrusion recovery.

Third, the number of recovery actions (column two of Table 5) compared to the number of legitimate actions (Snapshot columns) indicates that attacks often have a small footprint and hence the selective recovery approach is beneficial. However, more experience with real-world attacks is needed to validate this result.

Finally, the most significant result is that no policy performs ideally under all circumstances, a result that may seem undesirable. However, note that the optimistic policies have relatively few false positives while the conservative policies have few false negatives.

Policy	Propagation day	Recovery day	Propagation week	Recovery week
Snapshot	5.5±1.1	1.7±0.6	102.9±7.4	32.2±8.7
NoI	4.5±0.9	0.3±0.3	58.3±3.3	0.9±1.3
NoIAN	4.4±1.0	0.3±0.3	56.2±3.1	0.8±1.2
NoIANC	4.0±0.9	0.4±0.3	63.6±12.6	0.9±1.3

The performance numbers show the time to perform propagation and recovery one day and one week after the attack. Each number is shown in minutes and averaged across the different scenarios.

Table 6: Recovery performance

For example, the NoIANC policy had no false positives in the scenarios described above, while the NoI policy had no false negatives. For the experiments, we compared the difference in the outputs of the various policies and quickly determined the correct set of recovery actions. Recovery actions that occur in all policies are most likely to be correct actions. The difference in the outputs of these policies are the recovery actions that are ambiguous, i.e. whether these actions are tainted or legitimate. This difference is exactly the sum of false positives and negatives, which in our evaluation, is a small number. Hence, it is easy to classify these actions manually.

5.4 Performance Evaluation

In this section, we evaluate the viability of Taser by measuring the time to perform analysis and recovery and the performance and space overheads of auditing.

5.4.1 Recovery Time

Table 6 shows the time required to perform recovery when recovery is started one day and one week after the attack. The table shows the average recovery times and the 95% confidence intervals in minutes for each of the four analysis policies. The average is taken across all the scenarios evaluated in Section 5.3.

The trace phase was run once for each scenario. The tracing time for all scenarios is less than a minute and not shown in the table. The total propagation and recovery times are below 5 minutes when recovery is started one day after the attack. However, the propagation time can be more than one hour when recovery is performed after a week. The propagation time depends mainly on the time to recovery because propagation creates dependencies for every operation in the system. Averaging the recovery times across scenarios is reasonable because the propagation time dominates the total recovery time. The recovery time is typically short because it depends only on the number of objects that were tainted by the propagation phase. With the Snapshot policy, Taser needs to recover a much larger number of objects and hence the recovery times are larger.

The propagation phase is implemented in two steps. First, it queries the Forensix database to retrieve all the operations. Second, it creates dependencies between the operations. We found that approximately 80% of the propagation time is spent in querying the MySQL database. Although the numbers in Table 6 show the total propagation time, if all policies are run for a scenario, then the database query needs to be done once only. While storing the Forensix audit data in a database is useful for intrusion analysis, we believe that Taser can be greatly optimized by using an implementation specially designed for recovery.

While the performance results above show that the analysis and recovery phases can be run relatively quickly, note that the administrator must still spend time choosing the appropriate detection points or attack source objects between the trace and the propagation phases of the analyzer. Our entire recovery method depends on choosing correct detection points. Interactive and graphical analysis tools that can display the tainted source objects together with their attributes, such as the number of dependent objects, can ease this process.

Number of operations	13.3 Million
Size of events in flat file	1.9 GB
Size of database	2.3 GB
Database loading time	36.3 min

Table 7: Average daily backend statistics

5.4.2 Auditing Overhead

Auditing system call operations imposes overhead on the target system while loading the audit log into a database at the backend machine imposes overhead on the backend. With the load imposed by Galhogger on the Gallery photo album application, the logging overhead at the target is insignificant. The cost imposed at the backend, averaged per day, is shown in Table 7. The system load is stable and the daily numbers do not vary much over time. The table shows the number of system-call operations generated on the target machine. The most common operations consist of `read` (5.8 M), `open` (3.0 M), `close` (2.0 M), `mmap` (1.7M), `write` (371 K), `dup` (155 K), signal (106 K), `connection` (31 K), `unlink` (16 K), `exec` (9 K) and `fork` (8 K) events. These constitute 99% of all operations.

The total amount of uncompressed file data generated is 1.9 GB per day. When this data is loaded into a database, the database size grows by 2.3 GB per day. The loading time is 36.3 minutes per day. Another way to interpret this result is that the backend system can sustain loads that are approximately 40 $(24 * 60/36.3)$ times larger than the load imposed by Galhogger or one backend system can audit 40 target machines with the same load. The database loading time is the main bottleneck in Taser. An implementation optimized for Taser could potentially avoid the loading times.

Overhead Under Heavy Load. To measure the overhead of auditing at the target system under heavy loads, we ran the Webstone and the Linux kernel build benchmarks on the target system. The Webstone benchmark stresses a standard Apache web server running on the target by issuing back-to-back client requests and is representative of a loaded server environment. For the Webstone client, a third machine with the same configuration as the target machine was used and it was connected to the target machine with a Gigabit network. The kernel build benchmark is mainly CPU bound and determines the overhead imposed on the target system when running similar CPU bound applications in a regular desktop environment.

We measure the performance overhead of auditing on the target machine by logging the events on the target machine and streaming this data to the backend where it is stored in append-only files. Auditing has an insignificant effect on kernel compilation (0.6%) while Webstone throughput decreases from 258.3 Mbs to 239.2 Mbs or about 7.4%. These results are encouraging because they show that even under heavy load, the Forensix logging mechanism on the target has low overhead.

The amount of compressed data collected in these experiments (extrapolated per day) is 8 GB per day for kernel compilation and 11 GB per day for the Webstone benchmark. Even though the

amount of logged data is roughly the same, the Webstone throughput suffers more than kernel compilation because Webstone generated approximately 11 times the number of system call operations compared to kernel compilation. While the storage requirements of Taser can be large under heavy loads, we argue that the large amount of network capacity and massive and inexpensive storage space available in local networks today (e.g, a terrabyte costs between $500-$1000) make the Taser approach feasible and essential for reliably analyzing and recovering from intrusions.

6. RELATED WORK

Our work consists of three main components, analysis, recovery and auditing system activity. We focus on related work in these areas in turn. The analyzer is directly motivated by the work on backtracking intrusions [12]. This work uses a time-based approach to generate dependencies between processes, files and sockets and uses the dependency graph to view intrusions. The primary difference between the two systems stems from the difference in their goals. While backtracking is focused on tracking the sources of an intrusion, our analyzer generates a set of tainted files that need to be recovered. As a result, the backtracking taint analysis policies are conservative or else it would miss the intrusion, while we provide optimistic policies so that legitimate data can be preserved as much as possible during recovery. In addition, our optimistic policies use interval-based analysis.

Magpie [2] extracts the control flow and the resource requirements of requests in a clustered server environment by monitoring kernel and application-level events. Then it correlates these events using an application-specific event schema. Magpie uses interval-based correlation similar to our dependency intervals. However, while Magpie uses undirected dependencies to clustered sets of events, our analysis uses directed dependencies to derive data flow. Data lifetime analysis using system-level simulation [4] or hardware-based information flow [27] allows detecting or protecting programs against malicious attacks by identifying spurious information flows from untrusted I/O sources. Both can provide more accurate taint analysis than our approach but either run orders of magnitude times slower or require special architectural support.

Versioning file systems retain earlier versions of modified files, allowing recovery from user mistakes or system corruption. A key focus of versioning systems is encoding efficiency. For example, the Elephant file system [22] uses a clever purging method that keeps "landmark" data versions and purges generated and temporary files aggressively, while CVFS encodes metadata versions efficiently [26, 25]. Our system, which uses an unoptimized data storage mechanism, would benefit from some of these techniques, although purging data versions would limit some of the benefits of our recovery approach. While versioning approaches provide the basic capability to rollback system state to a previous time, such a rollback discards all modifications made since that time, regardless of whether they were done by a tainted or legitimate process.

The Repairable File System [33] has goals closest to our work. Its contamination analysis is similar to our taint analysis although it only uses a propagation phase and does not have any notion of analysis policies, intervals or white-lists. In addition, their file system does not seem to consider conflicting operations. Application-specific conflict resolution has been extensively studied in the context of replicated file systems [19, 13] and databases [29]. While we have not experimented with these policies, they would directly apply to our conflict resolution techniques.

Fastrek [17] recovers databases by attributing modifications to malicious activities and then rolling back changes selectively. A potential issue with this approach is cascading aborts where a legitimate operation is rolled back if it may have depended on the data produced by a tainted operation. While conservative analysis policies in our system effectively achieve the same result, our conflict resolution policies allow using optimistic policies that reduce this problem. Liu et al. [14] describe resolution algorithms that rewrite transaction history in a database by moving the attacking transaction and all affected transactions after non-affected transactions.

Brown [3] describes a recovery service that deals with operator errors in a mail server. Their system provides application-specific recovery that works well for a mail server, and while it is possible to extend the service to other applications, it is unclear how much effort is involved. In contrast, our system is geared towards server applications that do not necessarily have the well-defined semantics of a mail server and hence our techniques are more generic.

Sun [28] provides a safe execution environment (SEE) that enables users to try out new software (or configuration changes to existing software) without fear of damaging the system in any way. This is accomplished via a novel one-way isolation mechanism where processes running within the SEE are given read-access to the environment provided by the host OS, but their write operations do not affect the host until a commit point. The commit is performed if a consistency criteria is met or else the SEE is rolled back. This approach allows recovery only until the commit point. Furthermore, rollback caused by violating the consistency criteria can become more likely for long running SEEs.

Although our auditing system uses kernel-based logging it could, in principle, use other kernel auditing mechanisms such as VM-based auditing that can provide additional resistance to attacks on the logging mechanism [5, 7]. Sandboxing techniques are complementary to our approach. They interposition code that allows blocking program actions that may compromise security, while recovery deals with intrusions after they occur. Janus [9] interpositions system calls using the `proc` file system. Systrace [18] notifies the user about system calls executed by an application. Then it generates a sandboxing policy based on user response. Sandboxing raises the issue of policy selection, i.e, determining what actions are permissible for a given piece of software.

A large body of work has examined intrusion detection methods. Tripwire [11] monitors the cryptographic hash and size of key system files and directories and reports file accesses and modifications. SNORT [20] captures and logs network packets and detects intrusions based on predefined rules that match packet headers or data. Garfinkel [7] uses virtual machine monitor (VMM) based introspection to secure the detection mechanism from host intrusions.

7. CONCLUSIONS

Today, snapshot-based file-systems are typically used to recover from intrusions or human errors. This approach is well understood and easy to use but it works well only when intrusions or errors can be immediately detected. Otherwise, a snapshot before an attack loses legitimate user modifications that occur after the attack. We have described the design of the Taser intrusion recovery system that helps in recovering persistent data after an intrusion or local damage occurs. The key problems we address is determining the set of tainted file-system operations so that they can be reverted, and dealing with conflicts caused by dependencies between tainted and legitimate operations.

We evaluate the accuracy of Taser in dealing with a wide range of intrusions as well as erroneous user action scenarios. Our evaluation shows that our most optimistic analysis policy does not taint legitimate data but can miss intrusion activity, while the more conservative policies avoid missing any intrusion activity but require

some hand tuning using white list files. Our experience with Taser shows that an appropriate set of recovery actions can be determined quickly when the results of the different policies are compared. We believe that Taser provides the basis for developing automated intrusion recovery solutions.

In the future, we wish to explore whether Taser could be integrated with journaling and versioning file systems to improve scalability and to reduce disk space requirements. Taser currently does not support network file systems such as NFS since it audits at the client end which causes consistency issues for concurrent accesses. Implementing Taser at the server end would avoid this problem.

Acknowledgments

We greatly appreciate the valuable and detailed feedback from our shepherd Frans Kaashoek and the anonymous reviewers. The ideas in this paper were refined during several discussions with Wu-chang Feng, Jonathan Walpole, David Maier and Wu-chi Feng. We wish to thank Thomas Liu, Jim Snow, Lionel Litty, Alex Varshavsky, Borys Bradel and several other members of the SSRG group in Toronto who provided comments on initial drafts of the paper.

8. REFERENCES

[1] Edward C. Bailey. *Maximum RPM*. Sams, August 1997.

[2] Paul T. Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 259–272, 2004.

[3] Aaron B. Brown and David A. Patterson. Undo for operators: Building an undoable e-mail store. In *Proceedings of the USENIX Technical Conference*, pages 1–14, 2003.

[4] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the USENIX Security Symposium*, pages 321–336, August 2004.

[5] 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 USENIX Symposium on Operating Systems Design and Implementation*, December 2002.

[6] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proceedings of the Network and Distributed System Security Symposium*, February 2003.

[7] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Network and Distributed System Security Symposium*, February 2003.

[8] Ashvin Goel, Wu-chang Feng, David Maier, Wu-chi Feng, and Jonathan Walpole. Forensix: A robust, high-performance reconstruction system. In *Proceedings of the International Workshop on Security in Distributed Computing Systems (SDCS)*, June 2005. In conjunction with the International Conference on Distributed Computing Systems (ICDCS).

[9] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the USENIX Security Symposium*, 1996.

[10] Bobbie Harder. Microsoft windows system restore. `http://msdn.microsoft.com/library/en-us/dnwxp/html/windowsxpsystemrestore.asp`, April 2001.

[11] Gene H. Kim and Eugene H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 18–29, 1994.

[12] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the Symposium on Operating Systems Principles*, October 2003.

[13] Puneet Kumar and Mahadev Satyanarayanan. Flexible and safe resolution of file conflicts. In *Proceedings of the USENIX Technical Conference*, pages 95–106. USENIX, January 1995.

[14] Peng Liu, Paul Ammann, and Sushil Jajodia. Rewriting histories: Recovering from malicious transactions. *Distributed and Parallel Databases*, 8(1):7–40, 2000.

[15] Toby Miller. Analysis of the knark rootkit. `http://www.ossec.net/rootkits/studies/knark.txt`, 2001. SecurityFocus.

[16] Nicholas Petreley. Security report: Windows vs Linux. The Register, October 2004. `http://www.theregister.co.uk/security/security_report_windows_vs_linux`.

[17] Dhruv Pilania and Tzi cker Chiueh. Design, implementation, and evaluation of an intrusion resilient database system. Technical Report TR-124, SUNY, Stony Brook, April 2005.

[18] N. Provos. Improving host security with system call policies. In *Proceedings of the USENIX Security Symposium*, pages 257–272, August 2003.

[19] Peter Reiher, John S. Heidemann, David Ratner, Gregory Skinner, and Gerald J. Popek. Resolving file conflicts in the Ficus file system. In *USENIX Technical Conference*, pages 183–195. USENIX, June 1994.

[20] Martin Roesch. Snort - Lightweight intrusion detection for networks. In *Proceedings of the USENIX Large Installation Systems Administration Conference*, pages 229–238, November 1999.

[21] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.

[22] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the Symposium on Operating Systems Principles*, pages 110–123, December 1999.

[23] sd and devik. Linux on-the-fly kernel patching without LKM. Phrack issue 58, December 2001.

[24] Secunia. Secunia vulnerability report. `http://www.secunia.com`.

[25] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 43–58, 2003.

[26] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 165–180, 2000.

[27] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *ACM SIGARCH Computer Architecture News*, 32(5):85–96, 2004.

[28] Weiqing Sun, Zhenkai Liang, R. Sekar, and V.N. Venkatakrishnan. One-way Isolation: An Effective Approach for Realizing Safe Execution Environments. In *Proceedings of the Network and Distributed System Security Symposium*, February 2005.

[29] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 172–183, December 1995.

[30] Andy Watson and Paul Benn. Multiprotocol Data Access: NFS, CIFS, and HTTP. Technical Report TR3014, Network Appliance, Inc., 1999. `http://www.netapp.com/tech_library/3014.html`.

[31] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux Security Modules: General security support for the Linux kernel. In *Proceedings of the USENIX Security Symposium*, pages 17–31, 2002.

[32] Huagang Xie and et. al. Linux intrusion detection system (LIDS) project. `http://www.lids.org/`.

[33] Ningning Zhu and Tzi-Cker Chiueh. Design, implementation, and evaluation of repairable file service. In *Proceedings of the IEEE Dependable Systems and Networks*, pages 217–226, June 2003.