ROSIE - A RECOVERY-ORIENTED SECURITY SYSTEM

by

Shun Yee Chow

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Rosie - A Recovery-Oriented Security System

Shun Yee Chow

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2013

Recovery from intrusions is typically a very time-consuming and computationally expensive operation in current systems. One way to perform post-intrusion recovery is to use dependency-analysis. We observe that if an attacker can affect heavily-shared file-system objects on the target machine, such as the passwd file, then many other processes and files can be compromised from accessing this heavily-shared object. This in turn, would greatly increase the analysis time and recovery effort. Based on the notion that intrusions start with a network connection and then cascade into multiple system activities, such as file accesses and outgoing connections, we argue that the integrity of heavily-shared objects should be protected from the network, in order to minimize the time and effort required for recovery after an attack. In this thesis, we discuss our prototype Rosie, which is designed with incident response and post-intrusion recovery in mind. At its core, Rosie predicts how heavily-shared each file or process is, based on the previous system activities observed. For heavily-shared objects, Rosie enforces the appropriate mandatory access control and uses techniques such as sandboxing, in order to protect their integrity. In case that an attack is successful, Rosie still uses a tainting mechanism to track the information flow from the untrusted, network sources. The design of Rosie provides an important recovery guarantee, which is that after an attack has occurred, the maximum number of files that need to be recovered is at most equal to the dependency threshold, a value that can be adjusted by a system administrator.

Acknowledgments

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Despite our best efforts to build secure computer systems, intrusions and attacks are nearly impossible to avoid in practice. As users increasingly rely on the Internet for their daily computing needs, network-based attacks have become the most significant threat faced by host systems. Most existing solutions against network-based attacks rely on network-level technologies, such as the use of firewalls. Unfortunately, these solutions cannot effectively defend against host-based attacks because the root cause of the compromise is a vulnerability at the host, and network-level solutions often do not have sufficient visibility into host systems. A study performed by Mao et al. [33] has shown that the two key reasons why hosts are easily compromised are: 1) malicious and buggy software are common, and (2) the discretionary access control (DAC) mechanisms in today's operating systems are insufficient for preventing attacks.

A common way to address the weaknesses in DAC mechanisms is to provide mandatory access control (MAC) mechanisms in the operating system. At its core, mandatory access control applies a system-wide security policy that limits and restricts the access rights of processes. Examples of MAC-based systems include Janus [21], DTE Unix [8, 7], LOMAC [17], systrace [44], AppArmor [14] and Security Enhanced Linux (SELinux) [35]. Although these systems are powerful and can enhance the security protection offered by operating systems significantly, configuring them is often complex and an error-prone procedure. For example, it is commonly acknowledged that setting up SELinux is a difficult and daunting process [27], since it has over tens of different classes of objects, hundreds of possible operations, and thousands of policy rules for a typical system. An administrator needs to have a good understanding of these classes, operations, and policies in order to correctly design and specify the MAC policies for the programs and users on the system. Given the complexity of these systems, an administrator can make mistakes in the policy specification that can lead to hosts being attacked.

When a system is compromised, a user is typically forced to recover the machine by reinstalling the system and manually recovering the documents and system settings. Recovery usually involves rolling-back the system to a state prior to the attack, which is the most recent backup made before the attack. Even if the users create a complete backup of their machine daily, any legitimate changes made by the users since the attack are still lost. Since adversaries may actively hide their presence to avoid detection, an attack may be detected days or weeks after the system is compromised. In this case, using a old backup will result in days or weeks of lost work.

Our research group has previously designed a system called Taser [20] that aims to recover a compromised system while preserving a user's legitimate actions. Taser records all system-level activities during regular operation, and then once an attack is detected, it determines malicious activities by performing dependency analysis on the system calls in the recorded data. This analysis helps determine the attacking processes and all other files and processes that have been affected by the attacker. Then, Taser uses a log of file-system activities to enable reverting the persistent changes made by the attacker, thus restoring the system to a clean state. This approach needs to log all system-call activities since any activity could potentially be malicious. While such logging is not computationally intensive, it imposes heavy storage overhead, and analysis and recovery can take a long time because it involves analyzing all the system calls executed since the attack [29]. A second and more significant challenge with using dependency analysis for intrusion recovery is that if a heavily-shared file is affected by an attack, then the recovery effort can be significant. For example, if an adversary can affect the integrity of the password file, then any user that logs into the system after the compromise (which requires reading the potentially modified password file) is considered attack dependent or *tainted*, and all processes invoked by the user and the files modified these processes are considered tainted and thus reverted to their pre-attack versions.

In this work, we describe the design of a mandatory access control system called Rosie (recovery-oriented security) that limits the impact of compromises and enables recovery with minimal effort. The key idea behind Rosie is to use dependency analysis to rank files and processes (we refer to them collectively as *objects*) based on how many other objects they have affected in the past. Rosie marks all objects that have received any input from the network as tainted. Then it protects the integrity of a set of highly ranked objects by ensuring that these objects cannot be affected by tainted objects, which limits the total number of objects that can be affected by a network-based attacker.

Rosie protects highly ranked objects by ensuring two properties: 1) tainted processes cannot write data to highly ranked files, and 2) highly ranked processes cannot read data from tainted files. The challenge in the design lies in ensuring that these access control operations do not cause legitimate accesses to be denied. To reduce such false alarms, Rosie leverages an application-level virtualization system called Solitude [29] that uses a copy-on-write file-system for running untrusted applications. An untrusted process or its children running under Solitude can issue read operations directly to the underlying (or base) file system, but any write operations are confined to the copy-on-write file system. A user can manually commit writes, which applies the untrusted writes to the base file system.

The advantage of the Solitude transactional sandboxing mechanism is that the access control decisions for writes can be delayed until the commit. Traditional MAC based systems work by making an allow or deny decision when an access is attempted, which requires making the access control decision quickly and correctly. If these decisions are made incorrectly, e.g., a policy that allows a malicious process to write to a heavily-shared file, which then causes many other objects to become suspect, then recovery can become difficult. Instead, Rosie simply sandboxes all updates in the fast path, and then applies access control decisions periodically when committing objects. These decisions can be sophisticated, such as using our ranking method for determining heavily shared objects, because they can be run offline. In our approach, we commit untrusted files only if they do not affect the integrity of any highly ranked objects.

While Solitude was also designed to simplify post-intrusion recovery, it has three limitations. First, a user needs to manually run an untrusted application within Solitude. However, a typical user may not have enough knowledge of the trustworthiness of an application, so a user may mistakenly run a malicious application in the base system. Second, the user needs to manually commit sandboxed files. However, a regular user may not have enough experience to know whether an application has performed malicious updates, and could unwisely commit the updates to the base system. Third, although Solitude tracks dependencies created by committed files, it cannot limit how taints are propagated in the base file system after a file is committed. If a committed file can affect the content of some heavily-shared object, much more extensive, Taser-style recovery is needed.

The Rosie system makes three contributions. It addresses the limitations of Solitude by automating the decisions regarding 1) when to isolate untrusted applications, 2) when to commit untrusted files, and 3) how to limit potential attacks from committed files. First, Rosie tracks all processes that may have received information flow from the network and automatically sandboxes these processes using Solitude. Second, Rosie uses dependency analysis to predict how heavily-shared an object is likely to be, based on past file-system activities logged by the system. For each sandboxed file that may potentially be committed, Rosie commits the file if this action would not cause highly-ranked resources to become tainted. Otherwise, the files are considered unsafe to commit and remain in their sandboxes. Finally, Rosie ensures that all committed files, which based on our design, must have received information flow from the network, cannot pass on any information to highly-ranked processes. We achieve this property by simply denying all read accesses made by highly-ranked (i.e, trustworthy) processes to committed (i.e., tainted) files.

To evaluate our ideas, we have built a prototype of Rosie, and have run it on a server and a desktop machine running Ubuntu Linux 10.04.1 with three Intel Xeon CPU 3.00GHz processors, 2GB of RAM and a local ext3 hard disk. We have demonstrated that Rosie has a false positive rate between 0-0.52% for a server system, given that less than 10% of all file-system objects are heavily-shared (highly-ranked) in our test. Rosie also requires no application changes, incurs less than 720 Mb/day of storage on average, and requires very little user and administrator input.

Chapter 2

Related Work

In this section, we first provide an overview of the Solitude system [29] that we have used for sandboxing untrusted applications. Then, we discuss and compare other work related to our approach, including access control, virtualization and sandboxing techniques, file systems, and intrusion analysis and recovery.

2.1 Application-Level Sandboxing

In this chapter, we provide background on the Solitude system [29], which we use for sandboxing untrusted applications. Solitude is an application-level virtualization system that is designed to both limit the effects of attacks and simplify the post-intrusion recovery process. Figure 2.1 shows the architecture of Solitude. At its core, Solitude provides a file-system based isolation environment for running untrusted applications. Each isolation environment is bound to its own file-system namespace, similar to a process address space, via a copy-on-write isolation file system called IFS.

IFS offers a transparent view into the base (or regular) file-system for reading operations, but any modifications made by the untrusted process or its children processes are confined to the separate namespace. Programs that are run in IFS have restricted privileges to limit the effectiveness of malware such as spyware, rootkits, and memory-resident viruses that attempt

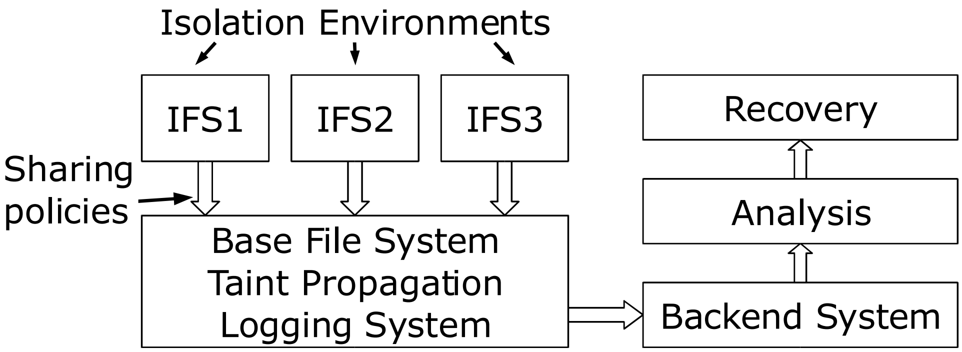

Figure 2.1: The Solitude Architecture

privileged operations (e.g., loading kernel modules), making it harder to compromise the IFS isolation mechanism. An example of using IFS is when an user downloads a photo editing program from a third-party website and is not totally convinced if the application is safe. So, the user invokes an IFS environment and installs the program within it. Running an untrusted application in IFS simplifies the recovery process. If a user (or administrator) decides that the application is malicious or is behaving undesirably, the user can simply discard the IFS, without worrying about the integrity of the base file-system.

As with any sandboxing method, there is a trade-off between the security provided by the IFS isolation environment, the application-level functionality, and the ease-of-use. The challenge arises when applications, running in different isolation environments, need to share data such as when a media file, downloaded in a browser, needs to be played by a media player. By design, sandboxed content in one IFS is not visible to other IFS's and the base file-system to prevent untrusted data from propagating throughout the entire system. When files need to be shared among different IFS's, Solitude allows explicit sharing of files between an isolation environment and the trusted base system (the explicit sharing needs to be enabled by the user via simple policies) via a *commit* operation. The commit operation synchronizes the base version of the file with the IFS version, but it marks the committed base file as tainted. Any use of these tainted files in the base is then logged and tracked by using our Taser system [20]. For example, if a committed file A is read by a base process B, which then writes to a base file C, both process B and file C would be marked as tainted as well. Processes running within IFS and the IFS versions of files are considered untrusted, but they are not tainted or the file operations logged, because we perform recovery of IFS files by discarding the entire IFS environment, rather than by performing recovery at a file granularity.

The benefit of using a commit-based scheme is that it delays synchronization with the base file system, allowing time to analyze whether an application is safe. Also, the source of tainting is limited to the explicitly shared files, so Solitude requires much less logging than full-recovery systems like Taser, which allow untrusted applications to modify the base file-system directly. Also, the reduced set of taint sources helps determine attack-related activities more accurately.

2.2 Access Control

Access control policies restrict access to a system and its objects based on a set of discretionary or mandatory policies. Discretionary policies permit users to entirely determine the access that is granted to the resources they own, allowing them to give access to these resources to other users, whether by accident or malice. For example, in the Unix context, a file owner may set file permissions incorrectly, allowing sharing of the user's files with other users. Mandatory policies are specified and centrally controlled by a security administrator, typically based on the principle of least privilege. Users do not have the ability to override the policy, e.g., grant access to files they can access to other users. As a result, mandatory policies are considered more secure and discussed in more detail below.

The Biba access control policies [9] are one of the earliest integrity protection models. With the Biba model, each subject and object is associated with an integrity label. An important and trusted subject (or object) has a higher integrity level than a subject (or object) that is labelled with lower integrity level. Biba includes five mandatory integrity policies:

1. **Strict integrity policy:** the subject and object integrity labels never change, and a subject can only read from objects with equal or higher integrity levels (no read-down) and write

only to objects with equal or lower integrity levels (no write-up).

2. **Subject low water-mark policy**: the subject can read any object, but its integrity level drops to that of the object after reading a lower-integrity object.

3. **Object low water-mark policy**: the subject can write to an object, but the object's integrity level drops to that of the subject after being written by a lower-integrity subject.

4. **Low-watermark integrity audit policy**: which combines (2) and (3) and freely allows any read or write to proceed, but drops the integrity levels of subjects and objects as needed.

5. **Ring policy**: which allows subjects to read lower-integrity objects while maintaining their integrity levels. Note that ring policy is applied to a subject; this implies that the subject is trusted to process lower-integrity information correctly.

Using Biba is challenging because the administrator needs to mark all objects as low or high integrity and then decide which policy to apply to each subject and object based on its desired usage. For example, LOMAC [17] is a prototype that implements the subject low water-mark policy. It is a dynamically loadable extension for commercial off-the-shelf (COTS) Unix kernels. In LOMAC, each object within the system is assigned an integrity level. The administrator can configure the system with a default policy but this policy is coarse-grained and uses simple rules to decide the integrity label for objects, without considering the actual usage of file system resources. Otherwise, LOMAC requires manual specification of a mapping between existing files and integrity levels for each installation, which can be a labour-intensive process. LOMAC also suffers from the self-revocation problem where an application can fail when its integrity level is lowered. For instance, suppose a subject at a high level creates an object, and subsequently observes another object, which is at a low level. According to the low water-mark model, this observation demotes the subject to the same low level. Consequently, the subject, now at a low level, cannot modify the object it created, which remains at a high level. With the LOMAC default policy, the vast majority of applications execute at the lowest level, and consequently cannot be demoted and are immune from the self-revocation problem. Applications that execute at higher levels are far fewer, and are mainly administrative in nature. However, the default policy provides no protection for subjects and objects at the lowest level, which implies that LOMAC can only provide limited security since a majority of objects are vulnerable to compromised subjects.

Another classic integrity-protection model is the Clark-Wilson model [13]. This model defines integrity as a set of integrity constraints, and the data is considered valid and consistent if it satisfies all of the constraints. The Clark-Wilson model also divides data items into constrained data items (CDIs) and unconstrained data items (UDIs). CDIs are considered to have high integrity, and can only be changed by transaction procedures (TPs) that are certified to change the CDIs in ways that preserve their integrity. Also, for each TP, the system needs to list which CDIs the TP is certified to access. Finally, the model also defines integrity verification procedures (IVPs), which test if the CDIs conform to the integrity constraints.

One of the main contributions of the Clark-Wilson model is that it enforces separation of duty with respect to the certified and the allowed relations. Specifically, it requires that the certifier of a transaction and the implementer be different entities, preventing problems such as the confused deputy [57]. However, in practice, the Clark-Wilson model is difficult to apply to an entire client or server system because it requires determining all possible system states. This model is more suitable when it is applied at the application layer, where the different program states are easier to define and list.

CW-Lite [48] is an improved integrity model that aims to preserve the information-flow rules of Clark-Wilson. CW-Lite requires the developer of the program to identify the program's inputs, and add annotations to the source code that indicate where input filtering should occur. Then, a system administrator uses CW-Lite to check whether low-integrity inputs are filtered properly, before they flow into high-integrity resources. CW-Lite detects and fixes integrity violations by building an information-flow graph, and then using a policy analysis tool called

Gokyo [26, 28] to find potentially illegal flows.

SELinux [35] is a modern mandatory access control model implemented and deployed in the Linux operating system. SELinux policies provide a powerful model for implementing least privilege because they can be fine grained. However, the size of the policies and and the many levels of indirection used (e.g., from programs to domains, then to types, and then to files) makes them difficult to understand. Furthermore, each new software installation requires updates to the policy to assign the appropriate labels to the newly added files and possibly add new domains and types. Therefore, it is commonly acknowledged that writing SELinux policies is a complicated process [27]. Other examples of mandatory access control systems include systrace [44] and LIDS [59], but they require extensive configuration expertise as well. In particular, both systrace and LIDS require intimate familiarity with UNIX internals.

Microsoft introduced a security feature in Vista called Mandatory Integrity Control (MIC) [40]. MIC partitions files and programs into four different integrity levels: low, medium, high, and system. Each process's integrity level is fixed to one of these levels. A program running at one level cannot update or delete objects that are at a higher level, or read or modify the memory of a process that has a higher integrity level than itself. However, MIC does not prevent a lower integrity process from sharing objects with a higher integrity process using shared objects, such as files, as long as the integrity level of the shared object is as low as the lower integrity process. This can trigger flaws in the higher integrity level process, and have it work on behalf of the lower integrity process, thereby causing a Squatting attack [45].

The most significant challenge in using the mandatory access control methods described previously is specifying the appropriate access control policies for a given system. Rosie provides a simpler model for specifying and implementing the access control policy. Instead of a single integrity label per object that needs to be manually specified, it uses two attributes, the importance level and the contamination level for each object. Our dependency-based ranking method indicates the importance of an object based on the recovery effort that would be required if the object were compromised, and a network-taint status helps determine the objects that may have been contaminated. The administrator only sets a dependency threshold to decide how many objects are considered important or highly ranked in the system.

Several recent systems make it easier to specify mandatory access control policies by either providing a learning mode or using information available from existing discretionary access control policies used in the system. For example, AppArmor [14] allows the system administrator to associate a security profile with each program that restricts the capabilities of that program. It includes a learning mode in which violations of the profile are logged but not prevented. This log can then be turned into a profile, based on the program's typical behavior, making it easier to specify the program's access policy. Rosie uses learning to determine the importance of object based on the dependencies it has formed previously. However, AppArmor does not maintain integrity levels for objects, and thus it cannot determine whether an object is contaminated, and it also cannot protect users from accidentally downloading and executing malicious programs.

The UMIP model [33] is an information-flow based model that, similar to Rosie, protects the system's integrity in the face of network-based attacks. This model leverages information available in existing discretionary access control policies to derive file labels for mandatory integrity protection. The basic UMIP policy partitions processes into low and high integrity. When a process performs an operation that can potentially contaminate itself, such as via reading from a network socket or communicating with another low integrity process, it drops its integrity and cannot perform sensitive operations. The basic UMIP policy enforces strict information flow, which can break existing applications and OS services. Hence, the basic UMIP policy needs to be enhanced with manually-specified capability exceptions to support server applications.

When Rosie is set to have a zero dependency threshold, Rosie and UMIP behaves similarly. For instance, all files will be treated as high-integrity and will be protected from tainted processes because they will be sandboxed. All the files within the system can be read by all processes, but they may only be modified by untainted (UMIP high-integrity) processes. This behaviour is comparable to UMIP, except that rather than denying write access from tainted (UMIP low-integrity) processes, Rosie would redirect the changes to the copy-on-write isolation file system. This approach is less likely to break applications because it still allows write-operations.

The UMIP researchers has also developed a model called Information Flow Enhanced Discretionary Access Control (IFEDAC) [38]. Similar to UMIP, IFEDAC combines discretionary access control (DAC) policies with dynamic information flow techniques. One of the weaknesses in current DAC systems is that it assumes a single user (or principal) is responsible for any request made to the file-system (e.g., a request to write to a file), whereas in reality the request may be influenced by multiple principals. Thus, the DAC mechanism cannot correctly identify the true origin(s) of a request and is susceptible to Trojan horses. To address this issue, IFEDAC maintains a set of principals (rather than a single one) for each process for each file system request. The request is only granted when every principal in the set is authorized to perform the request according to the DAC policy.

IFEDAC treats the DAC user account with login shell (but not root) and the remote network communication as the contamination sources. Each subject and object within the system has an integrity level, and its value is the set of contamination sources that indicate who may have gained control over the subject or who may have changed the content stored in the object. Like Rosie, IFEDAC uses information flow to track integrity levels. In addition, each object has a read protection class (rpc) and a write protection class (wpc). In most cases, the rpc and wpc contain the entities that are authorized to perform read and write, as determined by the DAC policy. When a subject requests to read (write) an object, the access is allowed if the subject's integrity level is a subset of the object's rpc (wpc), i.e., all of the contamination sources of the subject are allowed to access the object.

The integrity level and the wpc are analogous to the taint status and the importance level in Rosie. The key difference is in the way that these levels are specified. IFEDAC relies entirely on the existing DAC policy to determine which object is important and thus needs to be protected against contamination, whereas Rosie determines the importance of an object based on how many other files it has already influenced, and thus does not depend on a correct DAC specification. Unlike IFEDAC, currently Rosie does not consider DAC user accounts with login shell (but not root) as contamination sources. Incorporating strong user separation from IFEDAC would strengthen the Rosie security model.

SecGuard [2] is another integrity protection model that leverages information from the DAC policy. Similar to UMIP and IFEDAC, it uses existing DAC information to initialize integrity labels for subjects and objects within the system. In SecGuard, all subjects and objects have two integrity labels – important integrity and current integrity. Each integrity label can have 2 levels, either high and low. SecGuard maintains the two integrity levels for different purposes. Specifically, the current integrity label is mainly used for the mandatory access control policy of SecGuard, while the important integrity label is used to determine if the current integrity level can be promoted to high for exceptions.

All root (system-level) processes are regarded as important, and so their important integrity levels are set to high, and their current integrity levels are set to high as well during system boot. In comparison, normal (non system-level) processes would have their important integrity level and current integrity level set to low. When a process is created, it would inherit both the important integrity level and the current integrity level from its parent process. During a process' life cycle, its important integrity level cannot be changed, but its current integrity level can be changed dynamically according to the security policy in SecGuard. Specifically, the current integrity level of a subject will drop, if it 1) receives traffic from the remote network, 2) communicates with a low-integrity subject, or 3) reads a low-integrity object. To initialize the integrity labels for objects, SecGuard introduces a three-step initialization algorithm for objects in the system. First, it checks the other-bit in the 9-bits of DAC. If the file is world-writable, then its important and current integrity levels are set to low. If it is not world-writable, it checks all users in the object's user group. If there is an user whose important and current integrity levels are set to low, then the important and current integrity levels are set to low as well. Otherwise, it sets the current and integrity levels for the object as the same as its owner.

SecGuard shares the same challenges as UMIP and IFEDAC, where its strict access control can raise usability issues. To handle such situations, SecGuard provides partial trust subjects, which are essentially whitelisted subjects. Partially trusted subjects are treated as exceptions that can violate the security policies in SecGuard, so that their important integrity level and current integrity level can remain high after any read or write operation.

In many ways, the design principles among UMIP, IFEDAC, and IFEDAC are very similar. They all leverage the information from DAC to derive their integrity labels, and require exceptions of some sort to ensure the system remains usable. In comparison, most applications remain functional under Rosie because we run untrusted subjects in sandboxes, which reduces the number of subjects that need to be whitelisted. However, all of these three models provide a certain degree of confidentiality that is not handled in Rosie.

Sekar et al. developed a model called PPI [54] that automates the development of integrity labels and policies. PPI uses information-flow based techniques to protect the integrity of system files and processes, while aiming to preserve the usability of applications. To automate the generation of access control policies, PPI uses software package information (e.g., package contents and dependencies), a list of untrusted packages and/or files, a list of integrity-critical objects, and a log file that records resource accesses observed during normal operation on an unprotected system. The PPI model supports six policies: denying read, downgrading of subject, trust, denying write, downgrading of object, and redirect.

PPI is an automated system, so it is far more usable than other manual approaches for specifying the access control policy. Both PPI and Rosie use similar techniques to automatically derive access control policies for system files, based on past file-system activities. Hence, both systems can suffer from inaccuracy if the past history does not resemble the future behaviour of the target system. PPI and Rosie differ in their goals. While PPI aims to automatically derive the access control policy correctly, Rosie aims to reduce the recovery effort. For example, if the access patterns in a system change drastically over time, Rosie will automatically detect and update the importance of objects, while PPI would require an analyst to rerun the PPI algorithms.

Another type of access control model is functionality-based application confinement. Program access control lists (PACLs) [56], first proposed by Wichers et al., are likely the earliest form of application-oriented access control. PACLs label each file with a list of programs, specifying which programs can access it and the types of accesses that are allowed. Examples of systems that provide PACLs include the Tivoli Access Control Facility (TACF) [37] and the eTrust Access Control [1]. These controls typically protect particular resources only, rather than specifying all the resources available to applications. TOMOYO Linux [22] is another example that implements functionality-based application confinement. In TOMOYO Linux, an administrator needs to declare the behavior of each process, and the resources that the process needs to achieve its purpose. When protection is enabled, TOMOYO Linux enforces and restricts each process to their behaviour and resources. Unfortunately, functionality-based application confinement does not scale well to confine the large number of applications found on general-purpose computer systems. In addition, determining the access based on the user id and the program that is running can provided only limited protection. For example, consider a privileged program, such as insmod, with an effective root userid. If an attacker has compromised a daemon process that is running as root and has obtained a root shell, then the attacker may run the shell to run insmod, and install a kernel rootkit. In contrast, if insmod is started by a process that has never communicated or received information flow from the network, then the request would be from a local terminal and should be authorized. Therefore, we argue that Rosie's approach of treating all network-tainted processes as untrusted provides better protection against network attacks.

2.3 Virtualization

Applications can be isolated from each other by using common virtualization techniques such as hypervisor-based virtual machine isolation [3] and operating system-level virtualization [5]. Hypervisor-based virtual machines can provide strong isolation guarantees but they have limited support for sharing across virtual machines. For example, a virtual machine can be used to run multiple versions of the Office Word Processor in separate guest operating systems, but each guest needs to be administered separately, and has its own desktop, which may lead to a confusing and error-prone user experience.

With operating system-level virtualization, multiple isolated servers can be run on a single operating system. For example, multiple web server instances, using the same standard HTTP port 80, can be hosted on a single operating system. Each instance is called a container and the kernel provides resource management features to limit the impact of one container's activities on the other containers. Operating system-level virtualization is commonly used in virtual hosting environments, where it is useful for securely allocating finite hardware resources amongst a large number of mutually-distrusting users. This approach has been implemented in several operating systems such as BSD [30], Solaris [43], and Linux [49]. OS virtualization is designed primarily for isolating server applications that do not share any resources and have their own set of users and user directories. In contrast, we envision Rosie to also be suitable for desktop and client applications because it provides support for sharing files across its isolation environments.

Microsoft provides MS App-V [4] as their application-level virtualization solution. The MS App-V platform allows applications to be deployed in real-time to any client from a virtual application server. Instead of installing applications locally, only the App-V client needs to be installed on the client machine. App-V uses a single OS, but uses the SystemGuard virtual application environment [2] to keep application dependencies (DLLs, registry entries, fonts, etc.) separate from the rest of the system. This allows streaming and running multiple versions of an application (e.g., Office) within the same OS, but prevents the application from making changes to the client applications. Although SystemGuard uses a copy-on-write file-system, similar to Rosie, it does not allow users to commit applications or their configurations to the base, and also does not allow for auditing, tracking or recovery of the base system.

One-way Isolation [53] is a process isolation system that motivated the design of our Solitude [29] system. In One-way Isolation, untrusted processes can observe the environment of their host system, but the effects of these processes are isolated from the host system. Once an untrusted process is trusted, all changes made by it can be committed to the host system.

2.4 File Systems

Rosie uses a per-application namespace for running untrusted applications. The idea of per-process namespaces first appeared in Plan 9 [41], where it was motivated by representing various resources as file-systems.

Transactional file systems such as QuickSilver [23] and Microsoft's Transactional NTFS allow file system operations to be handled like transactions so that all the changes within a transaction are committed to disk atomically and the intermediate states of a transaction are not visible to other applications or transactions within the same application. Both QuickSilver and Transactional NTFS require changes to applications to use a transactional interface to start, abort, or commit a transaction, and they use a pessimistic locking mechanism for ensuring consistency. Quicksilver holds read locks on files until the file is closed, and write locks until the end of a transaction. Directories are locked when they are modified, for example when a directory is renamed, created or, deleted. The Transactional NTFS locking mechanism is also very similar to the QuickSilver mechanism. However, a file can be read and written in two different transactions concurrently. In this case, the reads do not see the modifications made by the other transactions. In contrast to these transactional file-systems, the IFS environment in Rosie supports existing applications without requiring any changes to the applications. It provides transactional semantics via commit operations, and hence transactions can exist for long periods of time. To ensure availability in the face of long-running transactions, IFS uses an optimistic concurrency control method that allows the different IFS environments to concurrently access and modify files. There are also a number of transactional file system prototypes for UNIX systems, including Valor [51], Amino [58], LFS [47], and transactional Ext3 file-system for TxOS [42], as well as transactional file-systems targeting embedded systems, such as TFFS [18].

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, in order to save storage space. For example, the Elephant file-system [46] uses an innovative method for purging generated and temporary files aggressively, while keeping landmark (i.e. the more important) data versions. Soules et al. have examined how to reduce the space requirements in Comprehensive Versioning File System (CVFS), by encoding the CVFS meta-data versions efficiently [50, 52]. Rosie has similarities with the Ventana file-system Ventana that provides sharing and file-level rollback with a rich file-system level versioning scheme. However, Ventana primarily focuses on using and managing virtual disks in a virtual machine environment, while Rosie aims to maintain integrity of highly-ranked objects to reduce the recovery effort required after an attack.

Compared to the versioning file-systems, checkpointing file systems such as Plan-9 [41], AFS [15], and WAFL [24], periodically generate efficient checkpoints of entire file systems for system rollback. Thus, checkpointing retains files at the granularity of the entire file system while versioning manages storage at the granularity of a file or groups of files. Currently, Rosie uses a non-optimized data storage mechanism for storing snapshots of committed files, and 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. Instead of purging data versions, we reduce memory and storage consumption by purging our dependency graph, which seems more appropriate for our recovery approach.

2.5 Intrusion Analysis and Recovery

Several efforts have focused on analysis and recovery of compromised systems. The Repairable File Service [60] logs file system activity and performs contamination analysis which helps with system recovery after an intrusion. The Backtracker [32] helps determine the source of attacks by tracking dependencies among kernel objects in reverse time order. By using a time-based approach, the Backtracker generates dependencies between processes, files, and sockets, and uses the dependency graph to view intrusions. In comparison, our previous system, Taser [20] determines and reverts the effects of malicious file-system activities by tracking similar dependencies in reverse and forward order.

Another intrusion detection and analysis system is ReVirt [16], which removes the reliance on the target operating system, by moving the target system into a virtual machine and logging below the virtual machine. This allows ReVirt to replay the system's execution before, during, and after an intruder compromises the system, even if the intruder replaces the target operating system. ReVirt logs sufficient information to replay the execution of a virtual machine at the instruction level. This enables it to provide detailed analysis about what occurred on the system, even in the presence of non-deterministic attacks and executions. Since many attacks exploit the unintended consequences of non-determinism (e.g., time-of-check to time-of-use race conditions [10]), ReVirt is a powerful tool that aids an administrator for replaying these types of non-deterministic attacks.

Hsu et al. [25] have proposed a malware removal framework that allows rolling back untrusted updates. Another powerful framework for application recovery is Operator Undo [11]. Operator Undo starts by intercepting and logging user interactions with a network service before they enter the system and creating a record of user intent. During an undo cycle, all system state is physically rewound using a storage checkpoint, and then the operator can perform arbitrary repairs. After the repairs are complete, lost user data is reintegrated into the repaired system by replaying the logged user interactions, while tracking and compensating for any resulting externally-visible inconsistencies. The authors use their system to recover from e-mail configuration bugs. The framework requires modifying applications to serialize requests that need to be replayed. It also requires separating persistent data and special recovery procedures for each type of application request. In comparison, Rosie can provide similar recovery functionality without modifying applications.

Polygraph [36] is a solution for recovering from data corruption in weakly consistent replicated storage systems. Like Rosie, Polygraph uses a tainting mechanism to track benign and potentially compromised data. However, unlike Rosie, Polygraph does not attempt to preserve legitimate changes to affected files.

Kim et al. have proposed a recovery system called Retro [31] that aims to repair a desktop or server after a compromise, by undoing the adversary's changes while preserving legitimate user's actions, with minimal user involvement and recovery effort. During normal operation, Retro records file system activities in an action history graph, which is essentially a dependency graph. During repair, Retro uses the action history graph to undo an unwanted action and its indirect effects by first rolling back its direct effects, and then re-executing legitimate actions that were influenced by that change. The way that Retro builds the action history graph and performs recovery through rollback and replay is very similar to Taser, Solitude, and Rosie's recovery approach. What is novel about Retro is how they reduce re-execution. To minimize user involvement during re-execution, Retro uses predicates to selectively re-execute only actions that were semantically affected by the adversary's changes, and uses compensating actions to handle external effects. To do so, Retro checks for equivalence of inputs to a process before and after repair, to decide whether re-executing the process is necessary. If the inputs to a process during repair are identical to the inputs originally seen by that process, Retro skips re-execution of that process. Thus, even if some of the files read by a process may have been changed during repair, Retro need not re-execute a process that did not read the changed parts of the file. They call this selective replay as retroactive patching.

Our research group has developed a data recovery system for web applications [6]. This system is a dependency-based recovery system for web applications, which is resilient to bugs and application misconfigurations. It tracks application requests, helping identify requests that cause data corruption, and reuses undo logs already kept by databases to selectively recover from the effects of these requests. In order to correlate requests across the multiple tiers of the application for determining the correct recovery actions, it uses dependencies both within and across requests at three layers (database, application, and client) to help identify data corruption accurately. Kim et al. have developed WARP, which uses retroactive patching to recover database-backed web applications [12]. WARP can be used to recover web applications from intrusions such as SQL injection, cross-site scripting, and click-jacking attacks, while preserving legitimate user changes, and can also be used to repair from configuration mistakes, such as accidentally giving permissions to the wrong user.

Although Retro, WARP and Rosie all aim to reduce recovery effort, their design for minimizing recovery work is different. Retro and WARP reduce the repair work by comparing the inputs during replay to avoid unnecessary re-execution, which requires recording precise dependencies in their action graph. Though this design may reduce the recovery time, it comes at a cost of significant storage overhead from recording such precise dependencies. For example, WARP incurs 2-3.2 GB/day storage overhead, while Rosie uses less than 700 MB/day. Also, if the input for an action comes from a non-deterministic operation, then Retro and WARP would still re-execute such action, even though the re-execution may not be necessary. Hence, Retro and WARP still suffer from false positives when selectively re-executing actions during recovery. In comparison, Rosie reduces the recovery effort by limiting the number of files that a tainted resource can possibly affect.

There are also several research efforts related to analysis and recovery for JavaScript attacks. For instance, Spectator [34] is a system used for detection and containment of JavaScript worms. Mugshot [39] is a system that captures every event in an executing JavaScript program, allowing developers to deterministically replay past executions of web applications. Mugshot's client-side component is implemented entirely in standard JavaScript, providing event capture on unmodified client browsers. However, since Mugshot only performs deterministic recording and replay of JavaScript events, it cannot replay events on a changed web page.

Chapter 3

Approach

This chapter describes the design of the Rosie system. Section 3.1 provide the motivation and an overview of the design. Section 3.2 describes the architecture of the target system that implements our ranking-based mandatory access control, while Section 3.3 describes the architecture of the backend system that runs an offline dependency algorithm for ranking objects. Finally, Section 3.4 analyzes the security provided by the system.

3.1 Overview

Our aim is to design of a mandatory access control system that limits the impact of compromises and thereby enables post-intrusion recovery with minimal effort. Rosie uses causality-based dependency analysis to rank file and process objects based on how many other objects they have affected in the past. Rosie marks all objects that have received any input from the network as tainted. Then it protects the integrity of a set of highly ranked objects by ensuring that these objects cannot be affected by tainted objects, which limits the total number of objects that can be affected by a network-based attacker. Rosie implements its access control by sandboxing untrusted applications using the copy-on-write file system based Solitude system (see Section 2.1 for a background on Solitude). We refer to trusted processes and files located in the base system as base processes and files, and untrusted processes running in Solitude and copy-on-write versions of files located in the IFS file system as Solitude (or IFS) processes and files. Next, we describe our method for ranking and tainting objects, and the Rosie access control policies that utilize the ranking and tainting information.

3.1.1 Ranking Objects

Rosie ranks base objects based on system-calls invoked on the base file system. Each base object is assigned a dependency value, defined as the number of *unique* base files that this object may have affected (i.e., the number of files that may have received information flow from this object). For instance, if a process P1 has written to file F1 and file F2, and process P2 writes to file F3 after reading from file F2, then process P1 would have a dependency value of 3 (it could have affected files F1, F2, and F3). When the dependency value of an object is greater than or equal to a value, defined by a system administrator, called *dependency threshold*, we consider the object to be heavily shared or highly ranked. Otherwise, the object is called lowly ranked.

We define the dependency value in terms of dependent files (and not processes) because post-intrusion recovery involves recovering the contents of files in our system. Note that this definition implies that when a process P reads from a file F, the dependency value does not change for any object. However, when a process P writes to the file F, the dependency value of other objects may increase if process P is dependent on them.

3.1.2 Tainting Objects

Based on the notion that intrusions start from a network connection and then cascade into multiple system activities, such as file accesses and outgoing connections, Rosie tracks which base objects have received information from the network by tainting these objects. For example, any base process that has read from a socket is marked as tainted, any base file written by a tainted process is marked as tainted, and any process that has read from a tainted file is marked as

Property	Description
High Process	Highly-ranked process does not become tainted
High File	Highly-ranked file does not become tainted
Tainted Object	Tainted object (file or process) does not become highly-ranked

Table 3.1: The Rosie Mandatory Access Control Properties

tainted. Also, all Solitude processes and files are considered untrusted and are thus implicitly tainted.

3.1.3 Ranking-Based Mandatory Access Control

The Rosie access control mechanism guarantees three properties, as shown in Table 3.1. The first two properties, called High Process and High File, ensure that highly ranked objects do not get tainted, and the third property, called Tainted Objects, ensures that tainted objects do not become highly ranked. The first two properties ensure that attacks cannot affect highly ranked objects, via processes and files. The last property provides the strong recovery guarantee that if a network-based attack compromises a Rosie system, then the maximum number of base files that needs to be examined and recovered during recovery would be at most equal to the dependency threshold. Specifically, a network-based attack can only affect tainted objects in the system, and if *none* of these objects (including the attacked process) can become highly ranked, then an attack can at most affect the threshold number of files. Setting the threshold value represents a tradeoff between the amount of recovery potentially required after an attack, and as we see later, the potential for applications to function incorrectly due to file accessed being denied.

Next, we describe a simple method for enforcing the properties shown in Table 3.1, which shows the limitations of this method. Then, we describe our approach for enforcing these properties. We consider any operation that reads a file or directory as a file read operation, and similarly for a file write operation. We can enforce the High Process property by ensuring that highly-ranked processes are not allowed to read data from tainted files or be spawned by tainted processes. Similarly, the High File property can be enforced by ensuring that tainted processes cannot write to highly ranked files. For the Tainted Objects property, objects can become highly ranked when their dependency value increases beyond the dependency threshold. This increase can occur when some process writes to a file, as explained in Section 3.1.1. Thus enforcing this property requires that whenever a tainted process writes to a file, the dependency value of all tainted objects that have affected this tainted process in the past be recalculated. If any of these tainted objects would become highly ranked, then this write must be disallowed.

There are two problems with this simple method of enforcing the properties shown in Table 3.1. First, denying read accesses (for enforcing High Process) or write accesses (for enforcing High File and Tainted Object) may raise false alarms and cause applications to malfunction. Second, it is expensive to calculate the dependency value for tainted objects on every write.

If we wish to enforce strong isolation and recovery guarantees, the only available option for enforcing the High Process property is to deny reads from tainted files. We expect that, in practice, highly ranked processes will not need to read from tainted files. However, we can avoid denying write accesses by using Solitude to sandbox the writes. This approach allows the application to write to Solitude files, while automatically ensuring that that the High File and Tainted Object properties are met since a base file is not being updated. Instead, these properties need to be enforced when Solitude files are committed to the base file system. A significant benefit of this sandboxing approach is that the dependency value calculation, needed for enforcing the Tainted Objects property, can be batched for all objects and performed periodically offline, before a commit is allowed, rather than on each write. In essence, our approach consists of: 1) using an isolation environment to sandbox untrusted processes, and 2) periodically committing file modifications made by these processes to the base file system, if these commits do not violate the recovery guarantee. Next, we describe this approach in more detail.

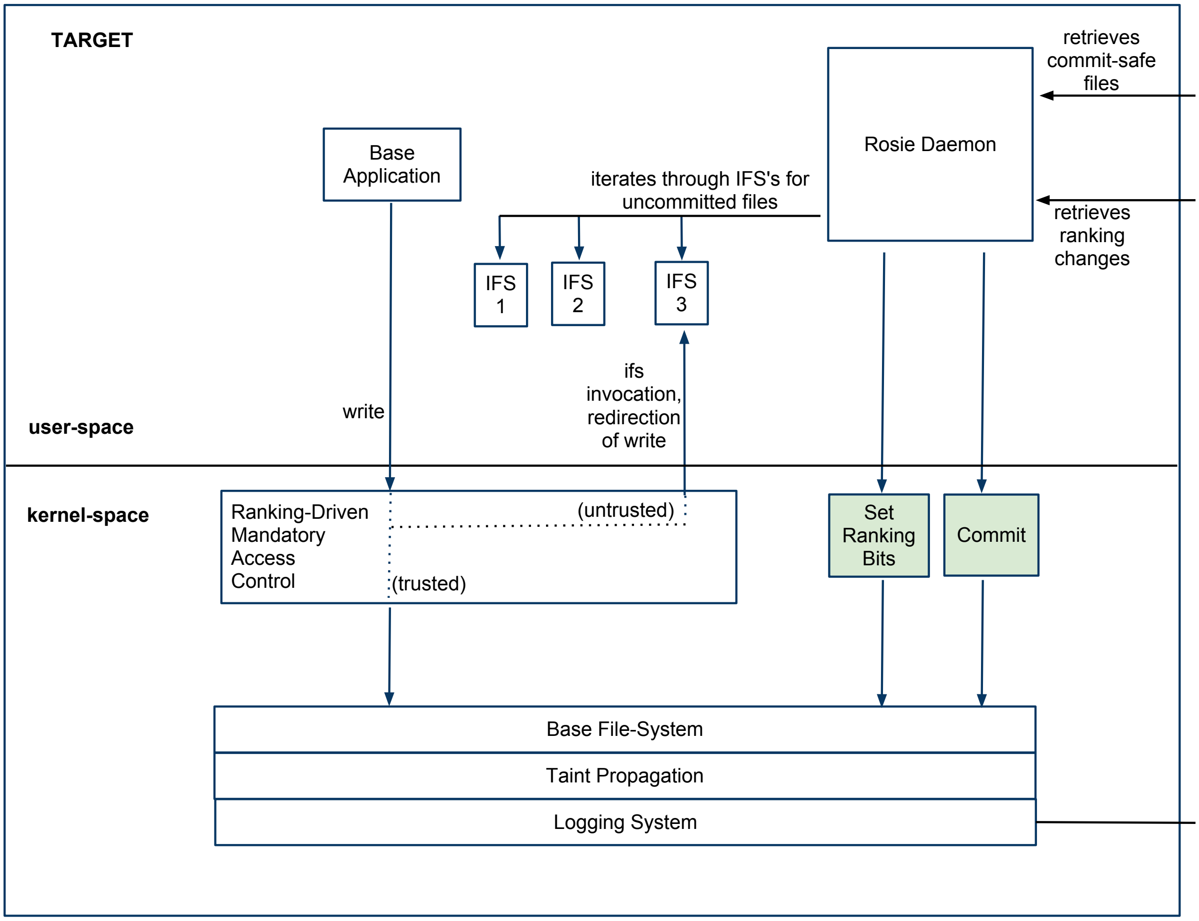
TARGET
retrieves commit-safe files
Rosie Daemon
Base Application
iterates through IFS's for uncommitted files
retrieves ranking changes
IFS 1
IFS 2
IFS 3
ifs invocation, redirection of write
write
user-space
kernel-space
Ranking-Driven Mandatory Access Control
(untrusted)
(trusted)
Set Ranking Bits
Commit
Base File-System
Taint Propagation
Logging System

Figure 3.1: The Rosie Architecture at the Target System

3.2 Target Operation

Figure 3.1 shows the architecture of Rosie on the target system. The target system runs the Solitude system, consisting of the base file system, taint propagation and the logging system in the kernel, and the IFS environments (see Section 2.1 also). The taint propagation system implements object tainting, as described in Section 3.1.2, by modifying the tainting implemented in Solitude, as discussed later in Section 4.1. The logging system logs read and write system calls issued by base processes and the taint status of base processes and files. This information is used by the backend for offline dependency analysis and for determining when Solitude files can be committed. We discuss the backend architecture later in Section 3.3.

Rosie adds two components to the target system, ranking-driven mandatory access control in the kernel, and the user-level Rosie daemon. The ranking-driven access control allows all read and write operations except for two cases. First, when a highly-ranked process attempts to read from a tainted file, the operation is denied, thus enforcing the High Process property. Second, when a tainted base process attempts to write to a base file, it invokes the IFS isolation environment and sandboxes the process, redirecting the write to a copy of the file in the isolation environment, thus enforcing the High File and Tainted Object property. The Rosie daemon communicates with the backend and performs two operations. It sets the ranking status of objects as highly ranked or not, and it commits files from the IFS environments to the base system. Next, we provide more details on these Rosie components by describing how the file read, write and commit operations are handled.

3.2.1 File Read

Figure 3.2 shows the flow chat for a file read operation on the target system. Similar to Solitude, Rosie allows a sandboxed process to read from the base file system transparently, unless a copy of this file already exists in the sandbox. In the latter case, the process must have previously attempted to write to the file, and so the file is copied to IFS (copy-on-write) and modifications are redirected to the IFS file. If a base process is highly-ranked, and the file is tainted, then the read operation is denied to ensure the High Process property. In all other cases, the read operation is allowed, and the logging system logs the read system-call. If the file being read is tainted, then this taint status propagates to the process and is logged.

3.2.2 File Write

Figure 3.3 shows the flow chart for a file write operation on the target system. Similar to Solitude, Rosie allows a sandboxed process to issue writes transparently, sandboxing them within the IFS file system. An untainted base process is allowed to write to a base file since this operation will not affect any of the Rosie properties shown in Table 3.1. In particular, a

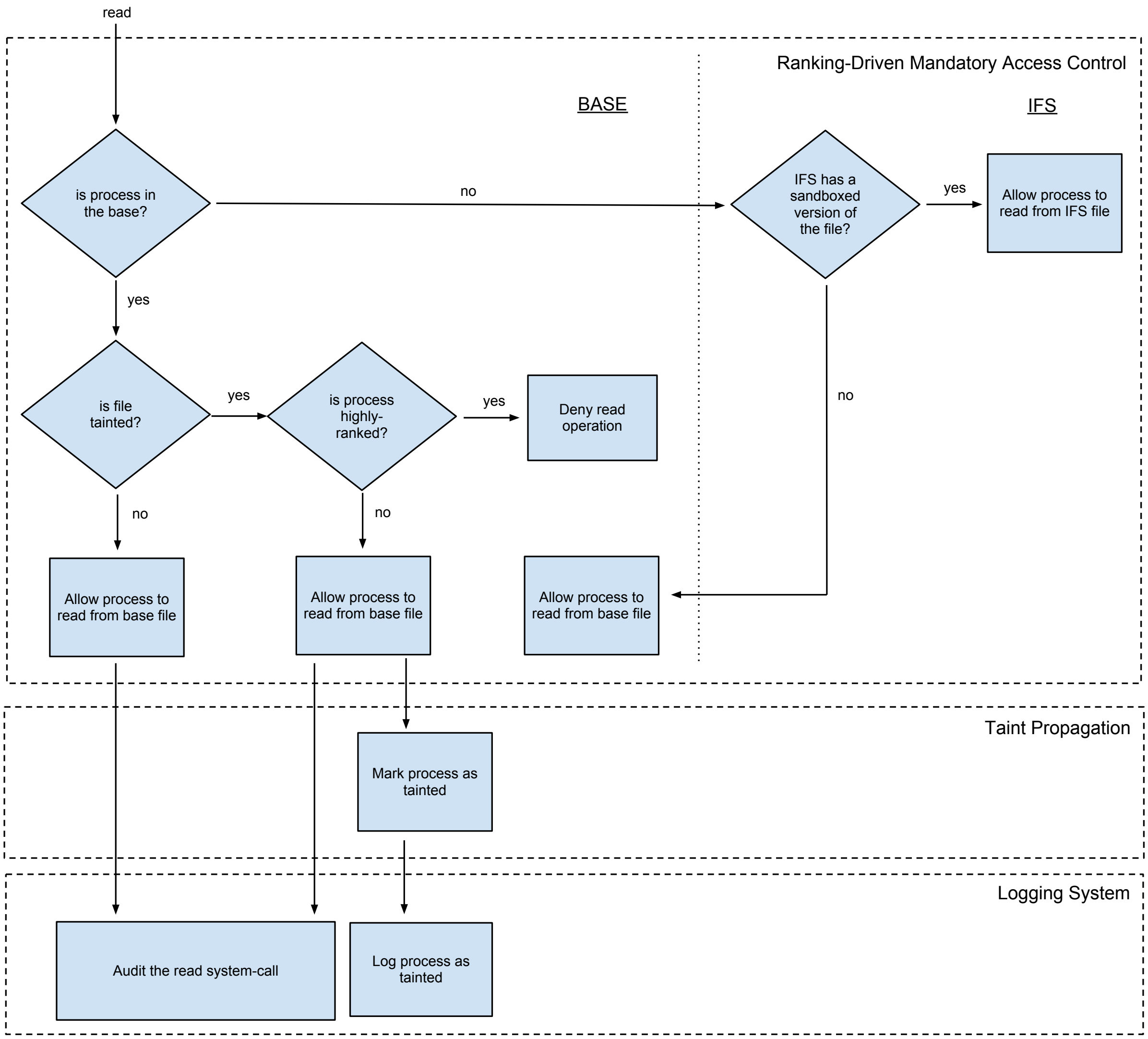
read
Ranking-Driven Mandatory Access Control
BASE
IFS
is process in the base?
no
IFS has a sandboxed version of the file?
yes
Allow process to read from IFS file
yes
is file tainted?
yes
is process highly-ranked?
yes
Deny read operation
no
no
no
Allow process to read from base file
Allow process to read from base file
Allow process to read from base file
Taint Propagation
Mark process as tainted
Logging System
Audit the read system-call
Log process as tainted

Figure 3.2: Read Operation

write operation never affects the High Process property, and it will not affect the High File property since a write by an untainted process will not taint an untainted file.

We show by contradiction that a write by an untainted process will not affect the Tainted Object property. We define an *ancestor* A of process P as a process or file that has causally affected P, i.e., there exists a path in the dependency graph from A to process P, and the edges in the path from A to process P have increasing timestamps. Suppose that the Tainted Object property is violated as a result of the write by process P. Then, the dependency value of some tainted ancestor A of process P must have crossed the dependency threshold. However, if the

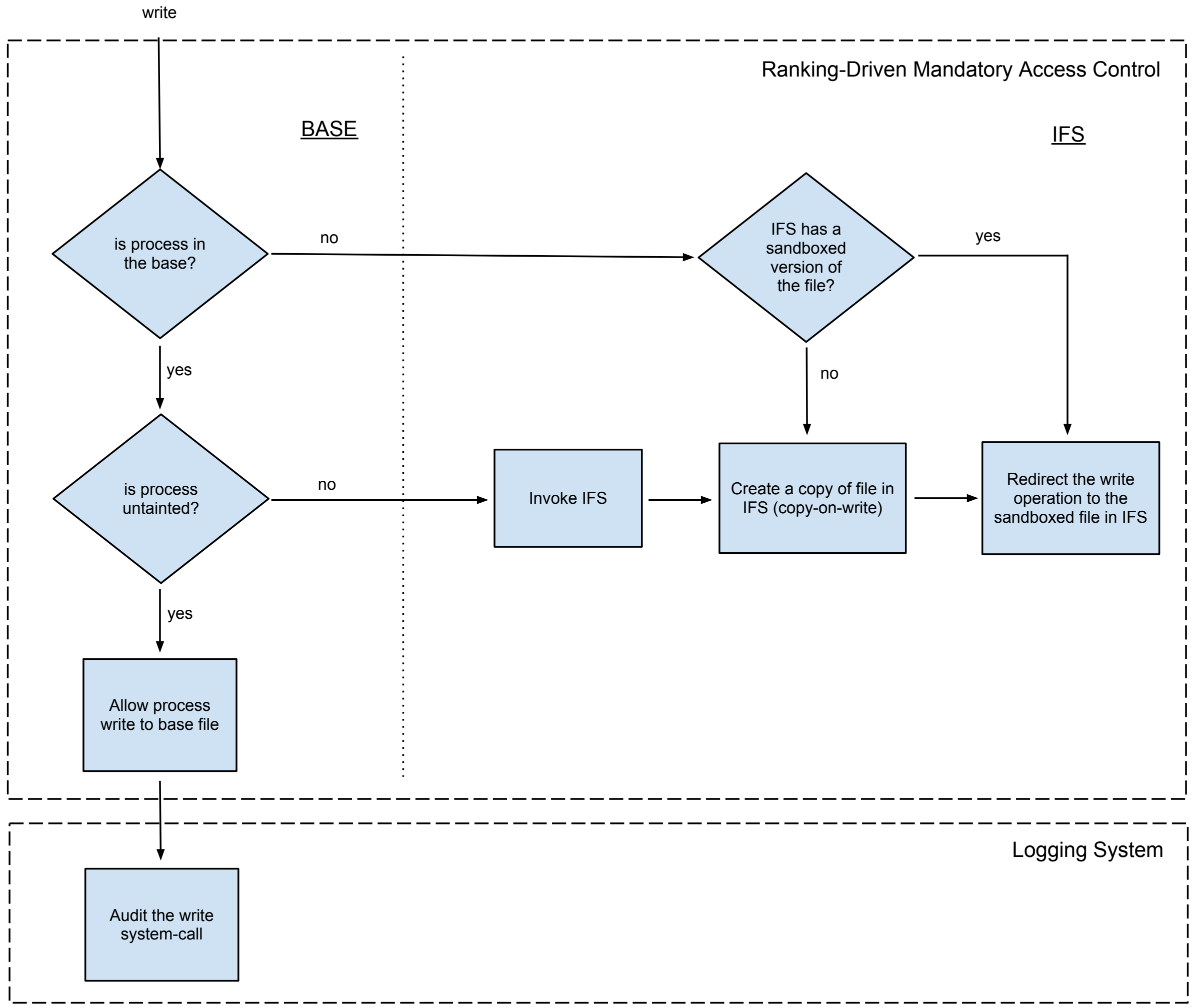
write
Ranking-Driven Mandatory Access Control
BASE
IFS
is process in the base?
no
IFS has a sandboxed version of the file?
yes
yes
no
is process untainted?
no
Invoke IFS
Create a copy of file in IFS (copy-on-write)
Redirect the write operation to the sandboxed file in IFS
yes
Allow process write to base file
Logging System
Audit the write system-call

Figure 3.3: Write Operation

ancestor A is tainted, then by the definition of tainting and an ancestor, the process P must be tainted (conversely, if a node is untainted, then all its ancestors must be untainted). Since process P is not tainted, the Tainted Object property cannot be violated.

The writes to the base file are logged to the backend system because they are needed for dependency analysis, as described in Sections 3.3.1 and 4.1.1. A tainted base process is sandboxed when it writes to any base file, as explained previously.

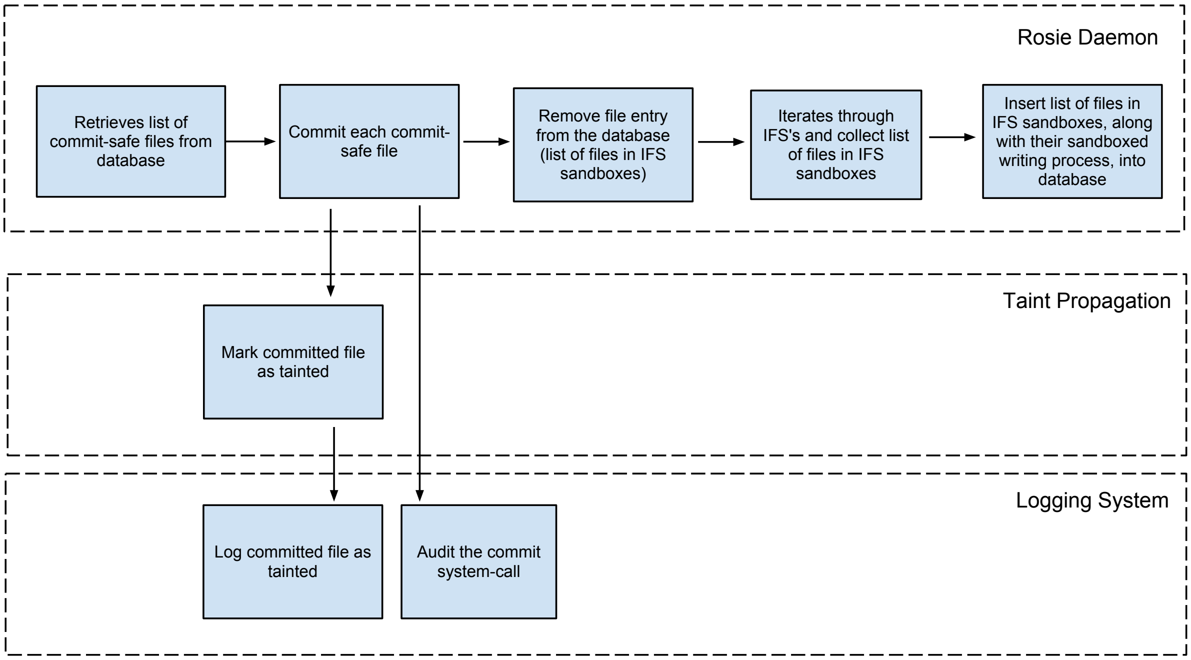
Rosie Daemon
Retrieves list of commit-safe files from database
Commit each commit-safe file
Remove file entry from the database (list of files in IFS sandboxes)
Iterates through IFS's and collect list of files in IFS sandboxes
Insert list of files in IFS sandboxes, along with their sandboxed writing process, into database
Taint Propagation
Mark committed file as tainted
Logging System
Log committed file as tainted
Audit the commit system-call

Figure 3.4: Commit Operation

3.2.3 File Commit

The commit operation periodically synchronizes the base version of the file with the IFS version and then removes the IFS version. A file commit that does not violate the High File and Tainted Object properties is called a *commit-safe* file. Only commit-safe files are committed, while the rest must remain within the IFS environment. The commit checker in the backend determines commit-safe files, as described later in Section 3.3.2.

Figure 3.4 shows the steps in a commit operation on the target system. First, the Rosie daemon retrieves the list of commit-safe files, determined by the commit checker before this commit cycle, from a backend database. Then the daemon commits the commit-safe files. The commit operation is atomic, being equivalent to an object being accessed and modified at the commit time, and hence we think of a commit as a deferred write operation to a base file. Each committed file is marked tainted and the system logs the commit system call, similar to logging the write system call. In addition, to ensure that recovery is possible in case malicious content is committed to the base, a snapshot of the base file is taken and sent to the backend before that file is overwritten by the commit operation. After committing all commit-safe files, their entries are deleted from the backend database. The Rosie daemon ensures that the commit-safe files are committed in the same order as they are analyzed by the commit checker, as explained in Section 3.3.2. Next, the Rosie daemon iterates through each IFS environment, collects the list of IFS files and the corresponding IFS process that created the IFS environment, and inserts entries for these files in the backend database. This list is used by the commit checker to generate commit-safe files for the next commit cycle.

3.3 Backend Operation

The backend performs offline dependency analysis for determining object dependency values and thus their ranking status, and uses a commit checker to generate a list of commit-safe files, as shown in Figure 3.5. These tasks are performed periodically by a user-level backend daemon process. Next, we describe dependency analysis and the commit checker in more detail.

3.3.1 Dependency Analysis

The dependency analysis calculation for ranking objects is run periodically, which amortizes its cost across all base file system operations that occurred within the period. Assume the start time of the first period is T0, the second period is T1, and the n^{th} period is Tn-1. The backend stores the system call logs, from start time T0, until the end time Tn. The dependency analysis builds a dependency graph and updates dependency values incrementally based on the logged system calls issued by base processes in the interval (Tn-1, Tn) in the n^{th} period, avoiding building and traversing the entire graph in each period. The graph nodes consists of base processes and files, with their dependency values and their taint status, and edges between these nodes represent read or write operations, labeled with the timestamp of the corresponding system call. An edge from a process to a file represents a write operation, and an edge from a file to a process represents a read operation.

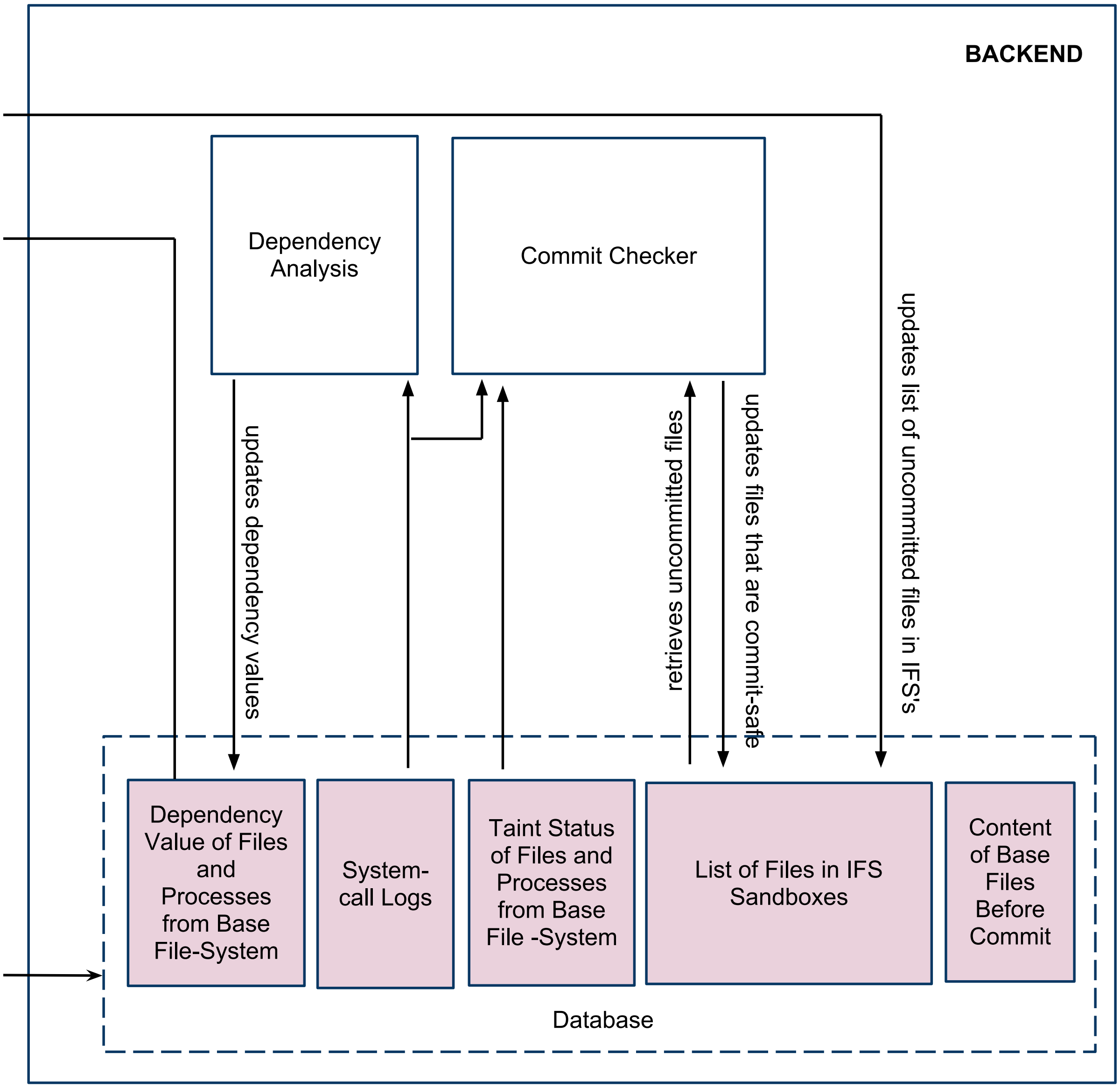
BACKEND
Dependency Analysis
Commit Checker
updates dependency values
retrieves uncommitted files
updates files that are commit-safe
updates list of uncommitted files in IFS's
Dependency Value of Files and Processes from Base File-System
System-call Logs
Taint Status of Files and Processes from Base File -System
List of Files in IFS Sandboxes
Content of Base Files Before Commit
Database

Figure 3.5: The Rosie Architecture at the Backend System

The object dependency values are updated as the graph is constructed. Recall that the dependency value of an object is the number of unique files that it has affected. This value is not affected by a read operation, as explained in Section 3.1.1. Now, suppose a process P writes to file F. For each ancestor A of process P (as defined in Section 3.2.2), if the file F was not previously dependent on A, then this write operation makes file F dependent on A, and the dependency value of A needs to be incremented by one.

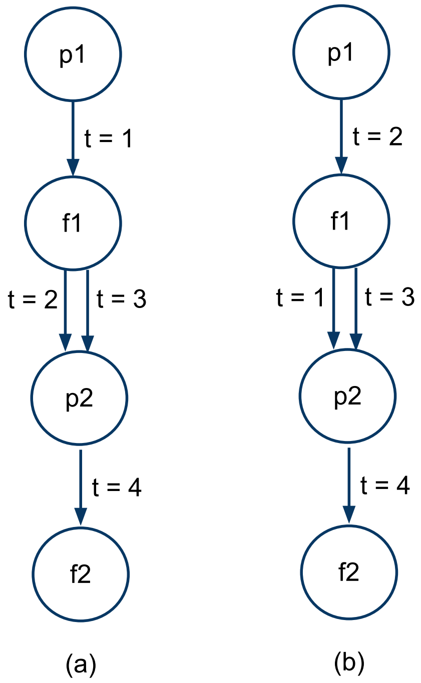

Figure 3.6: Removing Extra Edges Between Same Source and Destination

At the end of the analysis, the entire in-memory graph is stored in the backend database, and then loaded again at the start of the next period. To reduce analysis time and graph storage requirements, we next describe two optimizations, removal of extra edges, and removal of highly-ranked objects and their edges, that we have implemented in Rosie.

3.3.1.1 Removing Extra Edges

We have observed that some source objects have many edges to the destination objects. For instance, a browser may have cached an image file and it would read the file each time the site is loaded. Some of these edges can be removed without affecting the dependency value calculation. Consider Figure 3.6, which shows two graphs with multiple edges from the same source and destination nodes and the timestamps when these edges were created. In Figure 3.6(a), the edge from file F1 to process P1 at time t=3 can be removed from the graph without affecting the dependency value of process P1. In contrast, in Figure 3.6(b), if the same edge is removed, then the dependency between process P1 with file F2 would be lost. In general, we can remove

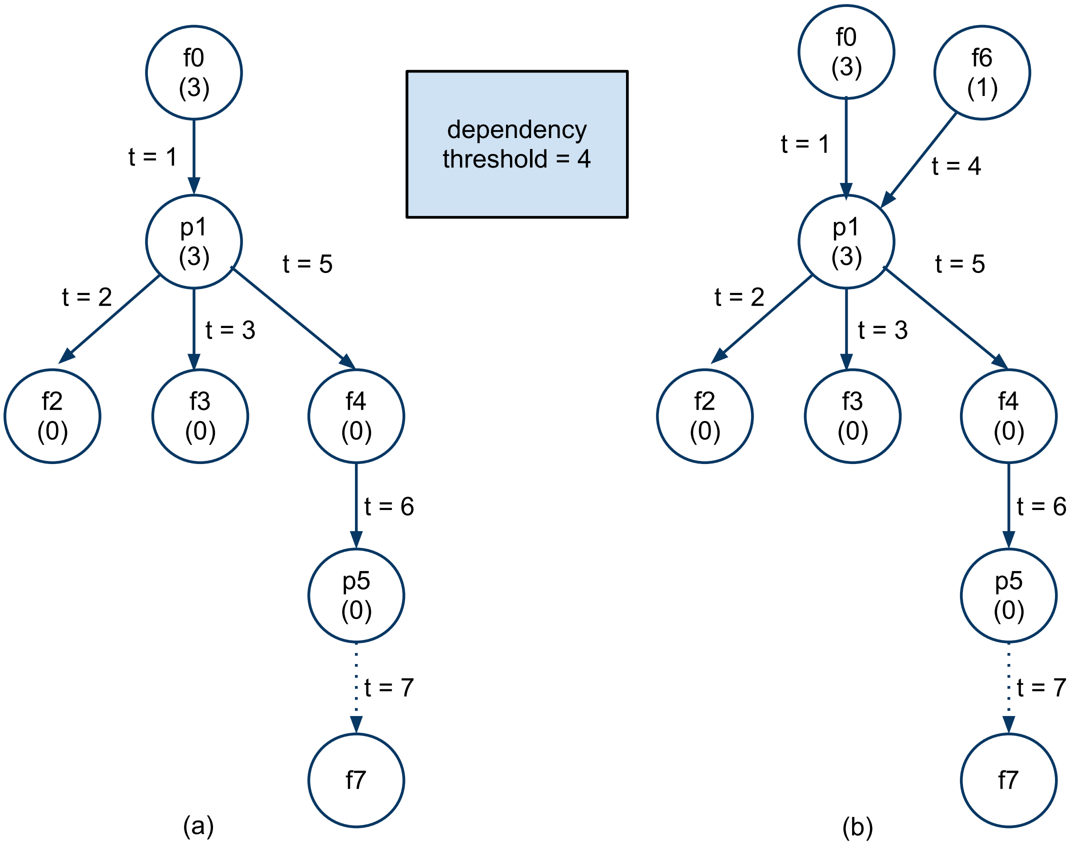

Figure 3.7: Removing Highly-Ranked Nodes

an edge, if there is another edge with the same source and destination with a smaller timestamp, and if there is no other intermediate edge pointing to the source node that has a timestamp between the two edges. We process system calls in timestamp order, adding earlier edges before the later edges, which greatly simplifies the implementation of this optimization.

3.3.1.2 Removing Highly-Ranked Objects

We can remove some highly ranked objects and their outgoing edges from the dependency graph because Rosie does not need the exact dependency value of highly-ranked objects. In particular, the mandatory access control mechanism needs to distinguish between highly ranked and lowly ranked objects, without knowing the exact dependency values. Similarly, the commit checker traverses the tainted and lowly-ranked objects in the dependency graph, as described in Section 3.3.2, but does not access the highly ranked files because they cannot be tainted by a commit.

Removing the highly ranked objects from the dependency graph reduces memory requirements, and the dependency values need to be updated for lowly ranked objects only. However, a highly-ranked object can only be removed from the graph, if and only if it is not dependent on any lowly-ranked object (such dependence is possible because of timestamp-based causal ordering), or else the dependency value of the lowly-ranked object could not be calculated correctly. Consider Figure 3.7 that shows two examples in which an object becomes highly ranked. In each of the figures, the dependency values of the nodes are listed in parenthesis. Assume that the dependency threshold is set to 4, and the figures show an edge being added to the graph when process P5 writes to file F7 at time t=7. After the edge is added, the dependency value of file F0 and process P1 would be updated to 4, and so they become highly ranked. In Figure 3.7(a), removing the highly-ranked nodes (F0 and P1) and their outbound edges does not affect the calculation of the dependency value for the lowly-ranked objects (F2, F3, F4, P5 and F7) in the graph. However, in Figure 3.7(b), process P1 is dependent on the lowly-ranked file F6, and so if file F0 and process P1 were removed, then the dependency between file F6 and its descendants (P4, P5 and F7) would be lost.

It is possible that after a node is removed from the graph, a future system call may access the removed node. In this case, this removed node is treated as a new node and is re-created in the graph. Although we remove highly-ranked nodes from the in-memory graph, we track them in the database. When the dependency analysis attempts to reinsert this node into the database at the end of the analysis, it does not do so since our algorithm notices that the same node already exist in the database and has a dependency value higher than the threshold. This check ensures that we do not replace the dependency value of a highly-ranked node with a value below the threshold. Note that this optimized algorithm does not guarantee the accuracy of the dependency value of highly-ranked objects, but it still correctly identifies highly-ranked objects.

3.3.1.3 Graph Purging

To reduce the dependency graph size and analysis time further, an administrator can fully rebuild the dependency graph, starting from a later start time. This allows us to ignore system call events that occurred before the new start time for ranking objects. Thus, rebuilding the graph can convert a highly ranked object to become a lowly ranked object. The ranking bit of these objects is updated by the Rosie daemon when it retrieves the changed ranking values of the base objects. After rebuilding the graph, we still maintain the original taint status of all base objects because the taint status and the dependency value are not related. Specifically, when a highly ranked object becomes lowly ranked, it does not violate any of the taint-related properties shown in Table 3.1. Rebuilding the dependency graph can also make files commit-safe. For instance, if a previously highly-ranked file has become lowly ranked, then it may be possible to commit this file. This may happen when a file was heavily accessed previously (in a causal sense) but has not been accessed much since the new start time.

3.3.2 Commit Checker

The commit checker runs periodically and determines whether IFS files are commit safe, i.e., they can be committed without violating the High File and Tainted Object properties. Recall from Section 3.2.3 that the Rosie daemon collects the list of IFS files and inserts entries for these files in the backend database. We call each such file a potential-commit file. The commit checker uses the in-memory dependency graph built by the dependency analysis, and simulates the commit operation for each potential commit file, one at a time, to determine whether the file is commit safe.

Recall that a commit operation is atomic, equivalent to an object being accessed and modified at the commit time. A commit is simulated by temporarily adding nodes corresponding to the sandboxed process and the base file that the commit operation would create or modify, and a directed edge between them representing the commit, to the graph. The commit checker ensures the High File property by simply checking that the base file is not highly ranked. It ensures the Tainted Object property by traversing the dependency graph in reverse order, looking for an ancestor (as defined in Section 3.2.2) of the potential-commit file that is tainted and has a dependency value that is one less than the dependency threshold (the traversal uses a depth-first algorithm, stopping at a node when none of its parents are tainted or have edges to the node with smaller timestamps). If such an ancestor is found, then committing the file will violate the Tainted Object property. In this case, the potential-commit file entry is marked as unsafe and the temporary nodes and the edge are removed. Otherwise, the file entry is marked safe, and the file commit is simulated by keeping the temporary nodes and the edge in the graph, and temporarily updating the graph dependency values using the dependency analysis algorithm. Then the next entry can be analyzed.

After all commit safe files are determined, the in-memory dependency graph, with the temporary updates, is no longer needed and it is discarded without storing it in the back-end database. The next time the dependency analysis is run, it uses the graph stored in the database before the commit checker was run (see Section 3.3.1). The Rosie daemon commits the commit-safe files in the same order as they were analyzed. This ensures that the Tainted Object property will still be preserved when the dependency graph gets updated as part of the normal dependency analysis after the commit operations.

3.3.3 Recovery

A system administrator performs intrusion recovery by using the tools developed in the Taser [20] and Solitude [29] systems. Recovery can be performed at a coarse-grained, per-sandbox granularity, or at a fine-grained, per-commit granularity. At a coarse-grained level, the entire isolation environment may simply be discarded during recovery, since the changes within the sandbox do not affect the integrity of the base file-system. However, if Rosie commits a malicious file to the base file-system, then the user or administrator needs to use dependency analysis to recover the system at the per-commit granularity.

As discussed previously in Section 3.2.3, the logging system sends information about each commit operation to the backend. This information includes a snapshot of the contents of the file before it becomes tainted, the time of the commit, and a Commit ID associated with this particular commit. As a starting point for performing recovery, the user or administrator may specify the isolated environment that this file was committed from and the associated Commit ID. We realize that the user or the administration may not have the knowledge of the commit operation, so she can use the analysis tools previously developed in Taser, to help determine the isolation environment and the rollback Commit ID. For example, if the file is known to be committed at some approximate time, then the closest previous Commit ID is chosen by the tool.

During fine-grained recovery, the set of files that depends on a particular commit ID are determined, and these files should be manually inspected to ensure the correctness of recovery. The Rosie properties guarantee that at most the dependency threshold number of base files need to be examined during recovery. Then, the files can be rolled back to an untainted state by using the unmodified selective redo algorithm in Taser [20]. Specifically, the rollback uses a snapshot of the file taken when the file was first tainted and then replays all subsequent legitimate modifications, as logged by the logging system.

3.4 Security Analysis

Rosie is designed to limit the impact of network-based attacks, rather than stopping or detecting them. After an attack, the maximum number of files that may need to be examined for recovery is the dependency threshold value. To implement this guarantee, consider what happens when a lowly-ranked object starts creating dependencies. If the object is not tainted, then it will eventually become highly-ranked, and then any accesses that may taint the object would be denied. If the object is tainted, then it will be sandboxed so that it does not become highly ranked.

Rosie implements its recovery guarantee by denying two types of accesses: 1) a highly ranked process cannot read data from tainted files, and 2) an IFS file cannot be committed to its corresponding base file if the base file is highly ranked or if the Tainted Object property may be violated. The first type of denied access may cause benign applications to fail, leading to a false alarm or a false positive. The second type of denied access has a more subtle problem. Since the IFS file cannot be committed, it remains within its IFS environment, and cannot be shared with other IFS environments. Furthermore, if the IFS is compromised, coarse-grained IFS-based recovery would lose the IFS version of the file. Setting a low dependency threshold reduces the amount of files that need to be examined during recovery but may cause more accesses to be denied, and vice versa.

Rosie uses object ranking for implementing is recovery guarantee. Object rankings are based on past file system activities and our system relies on future file system activities to resemble the past activities, or else accesses may be denied unnecessarily. For example, a highly ranked process cannot read tainted files, even though it may not affect many objects in the future. Similarly, a commit may be disallowed, even though the file may not affect many objects in the future.

Chapter 4

Implementation

Rosie is implemented as an extension of the Solitude implementation. Next, we discuss the Rosie implementation on the target and backend systems.

4.1 Target System

On the target system, Rosie implements object tainting and logging in the kernel by modifying the Solitude system, and it adds two components, ranking-driven mandatory access control in the kernel, and the Rosie daemon that runs at the user level, as shown in Figure 3.1. We discuss these components below.

4.1.1 Tainting and Logging

Solitude uses a tainting mechanism for intrusion analysis and recovery of the base system, as discussed in Chapter .2.1. It tracks files that have been committed to the base and objects that are affected by these committed files. For instance, if a base process has read a committed file, and then writes to another file, then this dependency is tracked using tainting. The tainting is implemented by associating a taint bit with each base file and process. The tainting is initiated by tainting a base file when its IFS version is first committed to the base system. The taint

Operation	**Taint Propagation Rules**
File and directory read operations, execute file	Tainted file→Taint process
File and directory modification operations	Tainted process→Taint file
Create child process	Tainted process→Taint child process

Table 4.1: Taint Propagation Rules

propagation rules are outlined in Table 4.1. Processes taint bits are not maintained persistently since processes are terminated when a machine is rebooted. File taint bits are maintained persistently across reboot.

We implement object tainting in Rosie, as described in Section 3.1.2, by simply changing the taint initiation rule. Since we are interested in network-based attacks, Rosie initiates tainting by setting the taint bit for processes that read from a network socket. The taint propagation rules in Rosie remain the same as in Solitude, as shown in Table 4.1.[1] With this change, when a network-tainted process writes to a file, it is automatically sandboxed, as shown in Figure 3.3, instead of users having to invoke Solitude sandboxes manually.

Solitude logs file system operations for intrusion recovery. When it is run on the target machine for the first time, a snapshot of the base file-system namespace (e.g., directory hierarchy) is sent to the backend, which allows recovering the namespace. Solitude logs when a file or process becomes tainted for the first time. Files become tainted for the first time when they are committed. At this point, a snapshot of the file's pre-tainted contents are sent to the backend system, which allows recovering the file to its pre-tainted contents. In addition, Solitude logs all tainted file operations, i.e., reads and writes to tainted files, as well as the contents of writes to tainted files, which helps with intrusion analysis and recovery using the Taser system [20]. Processes become tainted when they read from tainted files.

Rosie modifies the Solitude logging system by logging read and write operations to un-

[1] Our current implementation uses a separate tainting bit for the Solitude and the Rosie implementations for debugging purposes.

tainted files as well, as shown in the logging component in Figures 3.2 and 3.3. It needs to log all file operations because it uses this log to generate the dependency graph and the ranking status for all (untainted and tainted) files and processes.[2] However, similar to Solitude, Rosie does not log the contents of writes to untainted files, because untainted files are not affected by the network, and we assume that they do not require recovery. As described in Section 2.1, the IFS file operations are not logged (until files are committed to the base), because these operations do not affect the dependency values of base objects.

4.1.2 Ranking-Driven Mandatory Access Control

Rosie implements its mandatory access control policy during file-system related read and write operations as shown in the access control component in Figures 3.2 and 3.3. Solitude uses two bits per process or file, one for distinguishing between base and IFS processes, and the other for indicating its taint status. We have added a third bit to the Solitude implementation for indicating the ranking status of an object. As discussed earlier, Rosie only needs to track the taint and ranking status of base objects, and hence, we do not maintain these bits for sandboxed objects.

The most complicated part of the access control implementation is starting an IFS environment from within the kernel, when a tainted process in the base system attempts to write to a file, as shown by the `Invoke IFS` box in Figure 3.3. Solitude implements the IFS copy-on-write file system at the user level by using the File-system in Userspace (FUSE) framework [55]. FUSE allows virtual file system (VFS) operations to be redirected from the kernel to a user-level daemon that implements the user-level file system. Solitude provides a user level startup program that performs a series of setup and access control operations to create an IFS environment securely [29], before it runs an untrusted program in the IFS environment. Unlike Solitude, which creates untrusted processes in IFS, Rosie needs to migrate an existing tainted base process to an IFS environment when the process attempts to write to a file. To do so,

[2] In contrast, the commit checker only processes tainted objects.

Rosie uses a separate kernel thread that mimics the behavior of the Solitude startup program for creating an IFS environment. Then, it sets the IFS bit for the base process. Finally, it opens files in IFS that were open in the base process, and switches file descriptors so that the IFS process accesses IFS files instead of the base files.

4.1.3 Rosie Daemon

The Rosie daemon is responsible for periodically updating the ranking of base system objects and committing sandboxed files. It starts by retrieving updated object rankings and commit-safe files from the backend system, as shown in Figure 3.1 and discussed in Section 3.2.3. These rankings and the commit-safe files are generated when the backend daemon ran the dependency analysis and the commit checker last time, as discussed in Section 3.3. Based on the retrieved information, it updates the ranking bits, and commits all the commit-safe files to the base system. Finally, it sends an updated list of potential-commit IFS files to the backend system, as shown in Figure 3.5 and discussed in Section 3.3.2. For simplicity, the daemon performs all backend-related operations by directly accessing the database in the backend system.

While the daemon is performing its tasks, Rosie allows all file system read and write operations to proceed normally. However, these operations, which we call the intervening operations, introduce a window of vulnerability during which the the Rosie High Process and High File properties might be violated, because the ranking of an object is updated in a delayed manner on the target system. As a result, it is possible that an untainted process is currently marked as lowly ranked, and the intervening operations raise its rank (but this is determined later at the backend) and taint it. Similarly, an untainted file could be marked lowly ranked, and be considered commit-safe, but the intervening operations could raise its rank, and the commit would taint the file. These violations can occur for an object during a single period when its rank changes from low to high. However, the daemon can detect them at the next period when it updates the dependency analysis graph. We leave these objects marked as tainted and highly ranked. For example, when such a highly-ranked process reads a tainted file, the read is denied, and similarly, when the tainted process writes to a file, it is sandboxed. Similarly, a tainted and highly ranked file propagates its taint when it is read, and commits to the highly-ranked file are no longer safe.

In our current implementation, the worst case window of vulnerability is equal to the sum of the periods of the Rosie daemon and the backend daemon, because they run independently. If the Rosie daemon were run right after the backend daemon finishes its cycle, then the window of vulnerability would be close to the period of the backend daemon. Running the backend daemon more frequently would reduce the window of vulnerability further but increase performance overheads on the backend system.

The taint status of an object is set on the target system and hence intervening read and write operations will not violate the Tainted Object property (i.e., a tainted object will not be considered untainted and become highly ranked). This property is enforced by the commit checker and commits are performed under our control by the Rosie daemon.

4.2 Backend System

On the backend system, Rosie runs a daemon process that periodically performs dependency analysis for determining object dependency values and thus their ranking status, and runs the commit checker for generating a list of commit-safe files, as shown in Figure 3.5. We discuss these components below.

4.2.1 Dependency Analysis

The dependency analysis algorithm updates the dependency value of the ancestors of a process P, when the process writes to a file F, as described in Section 3.3.1. When the edge P→F already exists in the dependency graph, we traverse the graph twice, looking for ancestors of process P, before another P→F edge with the new timestamp is added, and after it is added. File F already depends on all ancestors found in the first traversal. Hence, we increment the dependency value by one for all new ancestors found in the second traversal. If the edge does not already exist, then we do not need to perform the first traversal. In the worst case scenario, this algorithm requires traversing the entire graph, with a complexity of O (|V| + |E|), where |V| represents the number of vertices on the graph, and |E| represents the number of edges on the graph, for each file write. However, we expect that in practice this traversal will require traversing a small number of nodes and edges.

The in-memory dependency graph is stored in the backend database after the analysis algorithm updates the dependency values. The nodes of the graph are stored in two database tables, `inode_dependency`, and `pid_dependency`, which represent the dependency values of files and processes respectively. The edges of the graph are stored in the database table `dependency_edge`, with each edge storing the source node, destination node, and the timestamp of when this dependency is formed.

4.2.2 Commit Checker

The commit checker runs periodically and determines whether potential-commit files, stored in a `potential_commit` database table in the backend system, are commit safe. For each potential-commit file, we simulate the commit operation, and then traverse the dependency graph looking for tainted ancestors of the file that may become highly ranked. If no such ancestor is found, the file is marked as commit-safe in the database table. Similar to the dependency algorithm, each potential-commit file check has a worst case complexity of O(|V| + |E|).

Chapter 5

Evaluation

Our evaluation of Rosie focuses on two aspects. First, we perform experiments to evaluate the trade-off between the security provided by Rosie and the usability of the system. Second, we evaluate the performance of Rosie, including its memory consumption, performance overhead, and the storage requirements.

5.1 False Positives

To evaluate the usability of the system, we measure how the dependency threshold effects the number of false positives in the system. We define a false positive as a file access that is denied during normal system operation. For processes, this occurs when a highly ranked process attempts to read a tainted file. For files, this occurs when an IFS file (which is tainted by definition) cannot be committed because its base version is highly ranked. [1]

One way to measure false positives is to use a trace of file system activities, and run Rosie on this trace, while tracking when a read or commit is denied, for different dependency thresholds. However, this evaluation method is expensive because it requires running a separate

[1] These false positive occur because a legitimate access is denied during normal system operation, in comparison with false positives in Taser [20] which are legitimate operations that are marked tained and reverted to a previous state during recovery.

dependency analysis for each dependency threshold. Also, when a read is denied, we do not know how the application would behave, and hence the trace would no longer be valid. Hecne, instead of varying the dependency threshold and measuring false positives for each dependency value, we estimate false positives on a system by running Rosie but without enforcing mandatory access control. Then we evaluate the dependency value of all tainted objects by running the dependency analysis algorithm. Any tainted object that is highly ranked for a given dependency threshold is considered a false positive since this access would have been disallowed by Rosie. This evaluation provides an upper bound on the number of false positives that would be generated in Rosie, because once an access is denied, it may avoid additional false positives. For example, if a highly-ranked process is not allowed to read a tainted file, it will not become tainted, and then if it writes to a highly ranked file, it will not spread its taint to the file. In our evaluation, the highly ranked file will become tainted, and will also be considered a false positive. The main benefit of our evaluation method is that we only need to run dependency analysis once to estimate false positives for any dependency value.

We perform this evaluation for an Ubuntu Linux 2.6.32 server system that provides several services, including web server (with a php/mysql backend), imap, webmail, postfix, NFS, dhcp, tftp, and sshd services, to a cluster of 128 machines, with roughly 10-15 active users. We use the Forensix system [19] for logging all file system activities on this server for six days (Jan 18, 2011-Jan 24, 2011), without any attack being involved. We use the first three days of logs (called *first set*) to identify all processes that have read data from network sockets. Then, we perform the dependency analysis on the second three days of logs (called *second set*). The first set allows us to taint processes in the second set that read data from a network socket before the dependency analysis is started on the second set.

Table 5.1 shows the statistics for all processes and files that were accessed on the server in the second set and the number of these objects that were tainted. Table 5.2 shows the number of highly ranked processes and files for different dependency thresholds. We have shown dependency threshold values that cause roughly a five percent increase in highly ranked

Total number of processes	12097
Total number of files	52359
Total number of objects	(12097+52359)=64456
Number of tainted processes	7701
Number of tainted files	1992
Total number of tainted objects	(7701+1992)=9693
Tainted objects/Total objects	(9693/64456)=15.04%

Table 5.1: Tainted Object Statistics

objects in the system, as shown in the last column of the table. Figure 5.1 shows the same data for highly ranked processes and files graphically, with the dependency threshold plotted on a log scale on the X axis.

Table 5.3 shows the number of number of false positives processes and files identified in the second set for different dependency threshold values. As explained earlier, these are objects that are tainted and highly ranked. We consider these objects as causing false positives because a highly ranked process would be denied access to a tainted file, and commits (which are tainted) would not be allowed to highly ranked files. Figure 5.2 shows the same data for false positive processes and files graphically, with the dependency threshold plotted on a log scale on the X axis.

Aside from the cases when all objects are highly-ranked (dependency threshold = 0) or lowly-ranked (dependency threshold = 64456), the results show that the false positives in this analysis range between 0.00%-2.68%. The result also shows that for our target system, when the dependency threshold is set to about 154, then only 638 out of 644656 objects are considered false positives (i.e. < 1% of false positives, which is an acceptable range). We expect that an administrator will run our analysis tool to set a dependency threshold that balances the usability and the security provided by the system. Tables 5.2 and 5.3 show that as the dependency threshold is lowered, the false positive rate increasely much more slowly than the

Dependency Threshold	Highly Ranked Processes	Highly Ranked Files	Highly Ranked Objects	% of Highly Ranked Objects
64456	0	0	0	0.00%
64336	0	2	2	0.001%
55777	0	8	8	0.01%
9993	1	11	12	0.1%
9809	49	603	652	1.00%
5434	235	2991	3226	5.00%
1510	283	6166	6399	10.01%
963	407	9263	9670	15.00%
649	575	12328	12903	20.02%
331	679	15464	16143	25.04%
322	679	15635	16314	25.31%
321	679	23229	23908	37.09%
313	679	25106	25785	40.00%
154	696	28314	29010	45.01%
15	1146	31117	32263	50.05%
2	1757	33619	35376	54.88%
1	2287	38380	40667	63.09%
0	12097	52359	64456	100.00%

Table 5.2: Highly Ranked Objects for Different Dependency Thresholds

percent of highly-ranked objects in the system. This shows that objects can become highly ranked but are often not tainted and hence protecting them will not cause false alarms during normal operation.

We have analyzed the files that show up as false positives even when the dependency threshold is high (few false positives). Table 5.4 lists these files. We are currently analyzing the

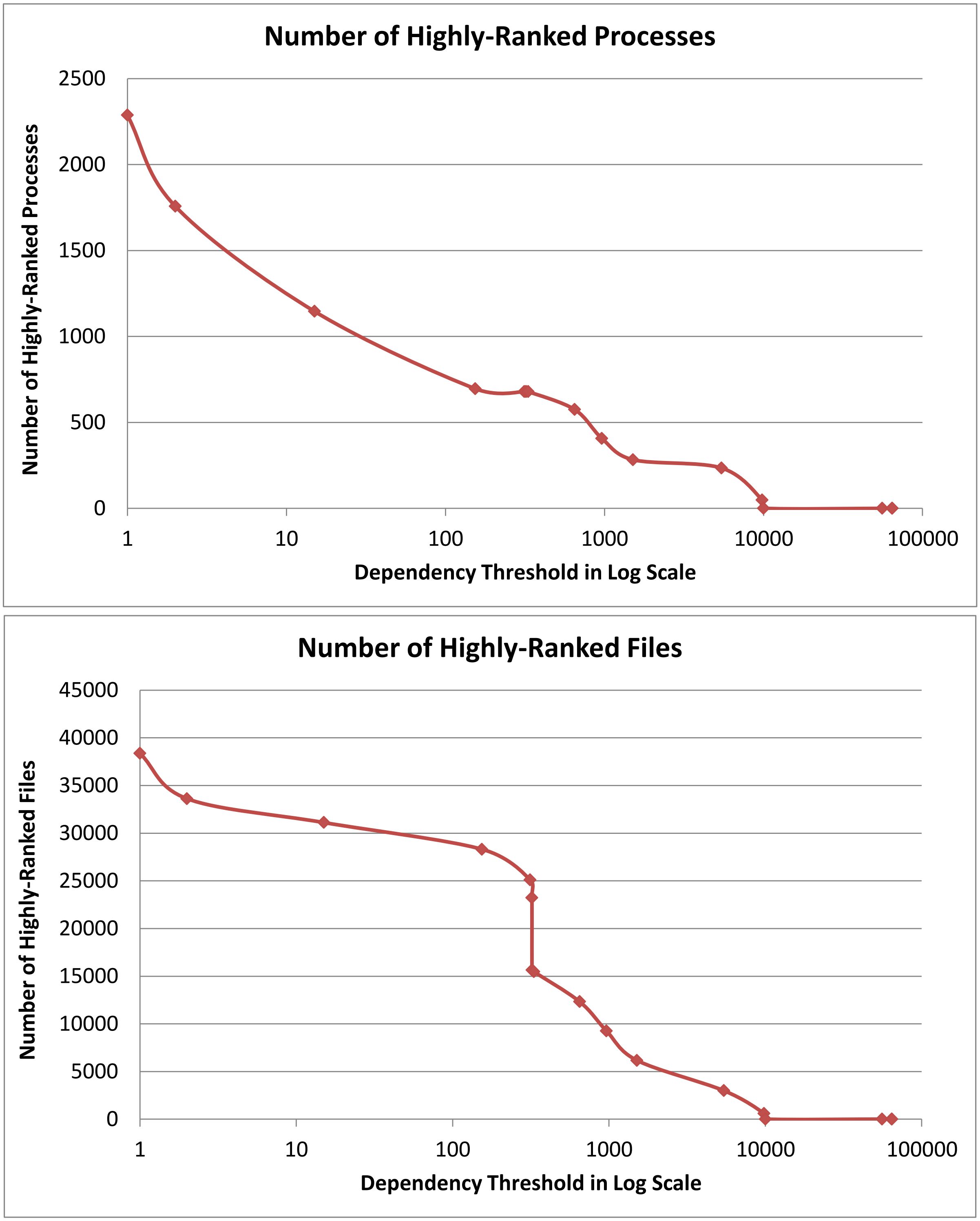

Figure 5.1: Highly-Ranked Processes and Files for Different Dependency Thresholds

Dependency Threshold	False Positive Processes	False Positive Files	False Positive Objects	% of False Positive Objects
64456	0	0	0	0.00%
64336	0	2	2	0.00%
55777	0	7	7	0.01%
9993	0	11	11	0.02%
9809	28	30	58	0.09%
5434	141	97	238	0.37%
1510	162	175	337	0.52%
963	233	191	424	0.66%
649	312	251	563	0.87%
331	365	266	631	0.98%
322	365	266	631	0.98%
321	365	266	631	0.98%
313	365	266	631	0.98%
154	370	268	638	0.99%
15	712	345	1057	1.64%
2	1009	393	1402	2.18%
1	1230	497	1727	2.68%
0	7701	1992	9693	15.04%

Table 5.3: False Positive Objects for Different Dependency Thresholds

reasons for all the false positives, whether a simple white listing approach can be used to avoid these false positives, and whether we can automatically adjust dependency thresholds based on training.

We limited our analysis period to three days due to memory constraints on the system running the dependency analysis. We performed the dependency analysis, which builds and

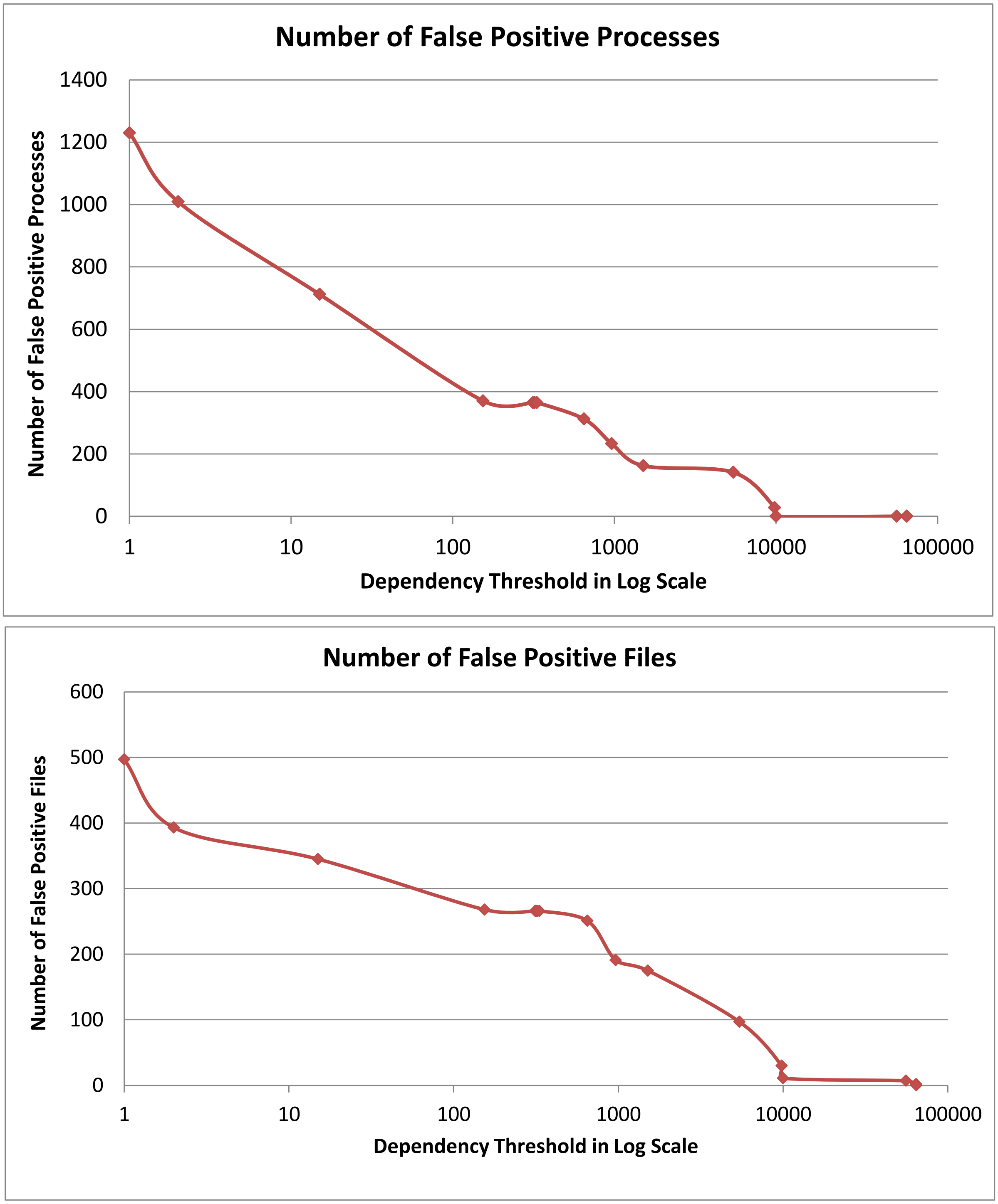

Figure 5.2: False Positive Processes and Files for Different Dependency Thresholds

traverses a dependency graph, on an Ubuntu Linux 10.04.1 machine with 2GB of memory. Each node of the dependency graph uses 48 bytes and each edge uses 52 bytes. We processed,

Path Name	Description
`/home/<user>/.bash_history`	Stores the history of commands typed by user on a terminal
`/proc/<pid>/oom_adj`	Under desperately low memory conditions, the out-of-memory (OOM) killer kicks picks a process to kill. *ooj_adj* acts as a knob, which stores a score (-17 to +15) of the process. The higher the score is, the more likely that the associated process is to be killed by OOM-killer.
`/var/run/motd`	Contains a "message of the day", used to send a common message to all users. The contents of the file /etc/motd are displayed by the Unix login command after a successful login, and just before it executes the login shell.
`/var/run/motd.new`	Since Ubuntu Intrepid Ibex (2008.10), Ubuntu has had a /etc/update-motd.d/ directory from which scripts are run to update the /etc/motd file. /var/run/motd.new is created by the scripts and then renamed to /var/run/motd.

Table 5.4: Common False Positive Files

on average, roughly two million system calls per day, and so our analysis algorithm runs out of memory after a few days.[2]

[2] The analysis removes extra edges (see Section 3.3.1.1), but does not remove highly-ranked objects so that the object rankings are obtained accurately (see Section 3.3.1.2).

5.2 Performance

In this section, we evaluate the performance, memory and space overheads imposed by our system. by measuring the time, the memory consumption, and the space overheads to perform various operations from Rosie. On the target system, we measure the overhead of our mandatory access control system. On the backend system, we evaluate the cost of running the dependency analysis algorithm and the commit checker.

The target system is a desktop machine running Ubuntu Linux 2.6.24-19 with two Intel(R) Pentium(R) 4 CPU 2.40GHz processors, 500 MB of RAM, and a local ext3 hard disk. The backend system runs Ubuntu Linux 10.04.1 with four Intel(R) Xeon(TM) CPU 3.00GHz processors, 2GB of RAM, and a local ext3 hard disk. The machines are connected by a Gigabit network.

5.2.1 Target System

We measured the performance overhead introduced by Rosie by running a set of workloads representing various client and server workloads Table 5.5 shows the running time for five different workloads on a native Linux system and on Rosie. Each workload was run five times and the results shown are the average of the five runs. In this evaluation, all these workloads are performed by a tainted process, which becomes sandboxed after its first file write.

The client workloads consist of: 1) untar of a Linux kernel source tarball, representing a file-system intensive workload, and 2) kernel build of the Linux source, which is mainly CPU bound and determines the overhead imposed when running similar CPU bound applications in a regular desktop environment. The server workloads consist of: 1) a large 200 MB file download, which stresses the file-system read performance, and represents a media streaming server, 2) a large 200 MB file upload, which stresses the file-system write performance, and represents an FTP or a video blogging site, and 3) the Apache ab benchmark, which stresses a standard Apache web server by issuing back-to-back requests using four concurrent processes

Workload	Time on Linux	Time on Rosie
Client - Untar, uncompress a kernel tarball	36.5s	169.7s
Client - Kernel build of Linux source	1244.3s	2162.5s
Server - Large 200 MB file download	18.5s	28.4s
Server - Large 200 MB file upload	18.4s	28.2s
Server - ab stress test	19.6ms	38.9ms

Table 5.5: Rosie Performance Overhead

running 20 clients that request files ranging from 1 KB to 15 KB, and is representative of a loaded server environment.

Out of the five different workloads, the Untar test introduces the largest performance overhead. The Untar test creates a large number files and directories, stressing the IFS isolation environment. Each of these files and directories invokes a copy-on-write operation, and each of these operations is redirected into the user-space code in our implementation. For the other workloads, Rosie has roughly 55-100% overhead. Much of the overhead in these workloads is caused by the time needed for setting up the isolation environbment, including the handling of open file descriptors, when setting up the environment.

5.2.2 Backend System

In this section, we measure the cost of running the dependency analysis algorithm, the commit checker, and the size of the database on the backend system.

5.2.2.1 Dependency Analysis

We measured the time to run the dependency analysis algorithm for the server dataset logged between Jan 18, 2011-Jan 24, 2011, as described in Section 5.1. Table 5.6 shows the size of the dependency graph and the time to build the graph for one day of file system activities, averaged over the six different days. The average number of system calls processed per day,

	Graph Size	**Graph Build Time**
Default implementation	118.4MB	213121s (59.2 hours)
Remove highly-ranked edges	99.0MB	37004s (10.3 hours)
Remove extra edges	50.6MB	12869s (3.6 hours)
Both optimizations	50.4MB	12822s (3.6 hours)

Table 5.6: Average Graph Size and Graph Build Time

over the 6 days, was 1,881,483. The default implementation takes more than a day to run for each day of data, making it clearly infeasible to use. We remove the highly-ranked edges by using a dependency threshold value that results in a false positive rate of less than 1% for the five runs (this value was 541). This optimization results in reducing the graph size by 16% and the build time by 83%. The extra edge removal optimization results in reducing the graph size by 57% and the build time by 94%, making it very effective. However, the optimization when combined, do not provide much benefit compared to removing the extra edges. Upon investigation, we found that many edges that are attached to the highly-ranked objects are also extra edges. We found that removing the extra edges is effective because the number of nodes in the graph increases slowly over time since many system calls are made on existing nodes, while the number of edges grows with the number of system calls, and a process invokes many system calls on the same file.

We have also measured the average time it takes to add an edge when updating the dependency graph. This time includes the time to create new nodes representing the source and destination of the system call (if they do not exist in the graph), creating and inserting the new edge, and updating the dependency value of the source node's ancestors. Table 5.7 shows these results for the second set of the server dataset logged between Jan 18, 2011-Jan 24, 2011, as described in Section 5.1. The results show the average time to add edges and the the average height traversed while updating the ancestors' dependency value, after the analysis has been running for different times (e.g., 12 hours, 24 hours, ..., 72 hours). These results do not include

Experiment Time	Nr. of System-Calls	Edge Add Time	Height Traversed
12 hours	491659	7.5ms	2.18
24 hours	949977	12.1ms	3.01
36 hours	1515464	18.9ms	3.44
48 hours	1904031	25.0ms	5.03
60 hours	3360359	45.8ms	6.30
72 hours	4505227	59.2ms	7.28

Table 5.7: Average Time to Add an Edge and Update Ancestors in the Dependency Graph

Average time to poll backend database	302.3ms
Average time to update object rankings	70.3ms
Number of changed object rankings	8.2

Table 5.8: Average Time to Update Object Rankings

the graph optimizations.

The Rosie daemon on the target machine polls the database backend periodically to get the changed object rankings and then sets these rankings on the target machine. Table 5.8 shows for each period, the time to poll the backend database, update the object rankings, and the number of objects whose ranking changes from low to high. These numbers are averaged over five intervals, with the daemon interval period set to two hours, and the dependency graph includes one day of system call data before the experiment is started. These numbers show that updating the object rankings takes a short time compared to calculating the rankings, justifying our offline dependency analysis approach.

5.2.2.2 Commit Checker

Table 5.9 shows the statistics for various commit related operations. These numbers are averaged over five intervals, with the daemon interval period set to two hours, and the dependency graph includes one day of system call data before the experiment is started. The Rosie dae-

Number of IFS environments	166
Time to iterate over IFS environments	1.1s
Number of potential-commit files	193
Time to log all potential-commit files	70.0ms
Time to check all potential-commit files	887.8ms
Number of commit-safe files	154.5
Time to commit all commit-safe files	15.8s

Table 5.9: Commit Operation Statistics

mon iterates over an average of 166 IFS environments and logs all the potential-commit files. Then the commit checker checks whether each of these files is commit safe. Finally, the Rosie daemon commits the commit-safe files. The commit of the commit-safe files dominates the entire process. This entire operation is run asynchronously without interfering with normal file system operation.

5.2.2.3 Database Size

In addition to the log of system calls issued on the target machine, Rosie also stores the taint status of objects, commit-related information (i.e. potential-commit entries, committed file entries), and dependency information in the database. Table 5.10 shows the size of the database for the second set of the server dataset. The various databases grow proportionally to the number of system calls. The dependency graph is updated using system calls in the most recent interval, and so loading this data and updating the dependency graph each interval should take time proportional to system activity.

Time	**System-Call DB Size**	**Dependency DB Size**	**Taint DB Size**	**Commit DB Size**	**Total DB Size**
24 hours	677.0MB	685.0MB	6.0MB	0.5MB	1368.5MB
48 hours	1420.5MB	1448.8MB	12.7MB	1.0MB	2883.0MB
72 hours	2127.9MB	2171.3MB	21.0MB	1.6MB	4321.8MB

Table 5.10: Rosie Database Size Statistics

Chapter 6

Conclusions

We have described Rosie, a recovery-oriented security system that is designed with incident response and post-intrusion recovery in mind. Rosie uses a ranking mechanism to predict the importance of a process or file object, based on the number of other files that have been causally affected by this object in the past. It uses a simple tainting mechanism to to keep track of the set of objects that could be compromised by a network-based attack. Rosie protects the integrity of important or highly ranked process and file objects by using a mandatory access control mechanism that ensures that highly ranked objects are not tainted, and tainted objects do not become highly ranked. This mechanism provides a strong guarantee that at most an administrator-defined threshold number of files can be affected by a network attack, thereby reducing the effort needed for recovery after an attack. Rosie implements its access control mechanism by building on a copy-on-write sandboxing environment called Solitude, previously developed in our research group.

There are several directions of future work related to this project. Currently, Rosie does not allow remote administration, since a remote process is automatically sandboxed and cannot directly modify the base system. We plan to investigate whether remote administration is feasible for a recovery-oriented security system. We plan to study whether our approach, based on automatically creating sandboxed applications and committing files, and different granularity recovery methods, are applicable to other systems, such as Windows, that provide several means of communication between applications, including registry entries. Currently, Rosie requires an administrator to select the dependency threshold manually. We plan to investigate techniques for determining the threshold automatically, so that the threshold is neither too low (causing read accesses to be denied) nor too high (an attack can affect many files). It may also be useful to set different thresholds for different groups of files and processes, based on their usage. Finally, we plan to explore ways to automate the recovery process, without relying on the administrator to identify the compromised files or the related commit operations.

Bibliography

[1] etrust access control for unix. Computer Associates, 2001.

[2] Microsoft softgrid. Microsoft Corporation, 2007. http://www.microsoft.com/systemcenter/softgrid/evaluation/virtualization.mspx.

[3] Hypervisor. Wikipedia, 2012. http://en.wikipedia.org/wiki/Hypervisor.

[4] Microsoft app-v. Wikipedia, 2012. http://en.wikipedia.org/wiki/Microsoft_App-V.

[5] Operating system-level virtualization. Wikipedia, 2012. http://en.wikipedia.org/wiki/Operating_system-level_virtualization.

[6] Istemi E. Akkus and Ashvin Goel. Data recovery for web applications. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2010.

[7] Lee Badger, Daniel F. Sterne, David L. Sherman, and Kenneth M. Walker. A domain and type enforcement UNIX prototype. *Computing Systems*, 9(1):47–83, 1996.

[8] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. Practical domain and type enforcement for UNIX. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1995.

[9] Kenneth J. Biba. Integrity considerations for secure computer systems. MTR 3153, MITRE, April 1977.

[10] Matt Bishop and Michael Dilger. Checking for race conditions on file accesses. *USENIX Computing Systems*, 9:131–152, 1996.

[11] 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, June 2003.

[12] Ramesh Chandra, Taesoo Kim, Meelap Shah, Neha Narula, and Nickolai Zeldovich. Intrusion recovery for database-backed web applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 101–114, 2011.

[13] David D. Clark and David R. Wilson. A comparision of commercial and military computer security policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 184–194, May 1987.

[14] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor. Subdomain:parsimonious server security. In *Proceedings of the 14th Conference on Systems Administration*, pages 355–368, 2000.

[15] James Lau Dave Hitz and Michael Malcolm. File system design for an nfs file server appliance. In *Proceedings of the 1994 Winter USENIX Technical Conference*, pages 235–245, January 1994.

[16] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 211–224, December 2002.

[17] Timonthy Fraser. Lomac: Low water-mark integrity protection for cots environments. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 230–245. IEEE Computer Society, 2000.

[18] Eran Gal and Sivan Toledo. A transactional flash file system for microcontrollers, 2005.

[19] Ashvin Goel, Wu chang Feng, Wu chi Feng, David Maier, and Jim Snow. Automatic high-performance reconstruction and recovery. *Journal of Computer Networks*, 51(5):1361–1377, April 2007.

[20] Ashvin Goel, Kenneth Po, Kamran Farhadiand, Zheng Li, and Eyalde Lara. The taser intrusion recovery system. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, October 2005.

[21] 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.

[22] Toshiharu Harada, Takashi Horie, and Kazuo Tanaka. Task oriented management obviates your onus on linux. Linux Conference, 2004.

[23] Roger Haskin, Yoni Malachi, Wayne Sawdon, and Gregory Chan. Recovery management in quicksilver. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 107–108, 1987.

[24] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6:51–81, February 1988.

[25] Francis Hsu, Hao Chen, Thomas Ristenpart, Jason Li, and Zhendong Su. Back to the future: A framework for automatic malware removal and system repair. In *Proceedings of the Annual Computer Security Applications Conference*, pages 223–236, December 2006.

[26] Trent Jaeger, Antony Edwards, and Xiaolan Zhang. Policy management using access control spaces. *ACM Transactions on Information and System Security (TISSEC)*, 6(3):327–364, August 2003.

[27] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Analyzing integrity protection in the selinux example policy. In *Proceedings of the USENIX Security Symposium*, pages 59–74, August 2003.

[28] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Analyzing integrity protection in the selinux example policy. In *Proceedings of the 12th conference on USENIX Security Symposium*, 2003.

[29] Shvetank Jain, Fareha Shafique, Vladan Djeric, and Ashvin Goel. Application-level isolation and recovery with solitude. In *Proceedings of the EuroSys conference*, April 2008.

[30] Poul-Henning Kamp and R.N.M. Watson. Jails: Confining the omnipotent root. In *Proceedings of the Second International SANE Conference*, 2002.

[31] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and Frans Kaashoek. Intrusion recovery using selective re-execution. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, pages 89–104, 2010.

[32] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 223–236, October 2003.

[33] Ninghui Li, Ziqing Mao, and Hong Chen. Usable mandatory integrity protection for operating systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 164–178, 2006.

[34] Benjamin Livshits and Weidong Cui. Spectator: Detection and containment of javascript worms. In *Proceedings of the 2008 USENIX Annual Technical Conference*, June 2008.

[35] Peter Loscocco and Stephen Smalleyr. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the Freenix Track of USENIX Technical Conference*, June 2001.

[36] Prince Mahajan, Ramakrishna Kotla, Catherine C. Marshall, Venugopalan Ramasubramanian, Thomas L. Rodeheffer, Douglas B. Terry, and Ted Wobber. Effective and efficient compromise recovery for weakly consistent replication. In *Proceedings of the ACM EuroSys Conference*, March 2009.

[37] Edson Manoel, Volker Budai, Axel Buecker, David Edwards, and Ardy Samson. Enterprise security management with tivoli. Technical Report 327, IBM Corporation, 2000.

[38] Ziqing Mao, Ninghui Li, Hong Chen, and Xuxian Jiang. Combining discretionary policy with mandatory information flow in operating systems. *ACM Transactions on Information and System Security*, 14(24), November 2011.

[39] James Mickens, Jeremy Elson, and Jon Howell. Mugshot: Deterministic capture and replay for javascript applications. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation*, April 2010.

[40] microsoft.com. The advantages of running applications on windows vista, 2007. `http://msdn2.microsoft.com/en-us/library/bb188739.aspx`.

[41] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in plan 9. *ACM Operating Systems Review*, 27(2):72–76, 1993.

[42] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, October 2009.

[43] Daniel Price and Andrew Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In *Proceedings of the USENIX Large Installation Systems Administration Conference*, 2004.

[44] Niels Provos. Improving host security with system call policies. In *Proceedings of the USENIX Security Symposium*, 2003.

[45] Mark Russinovich. Psexec, user account control and security boundaries, 2007. `http://blogs.technet.com/b/markrussinovich/archive/2007/02/12/638372.aspx`.

[46] 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 (SOSP)*, pages 110–123, December 1999.

[47] Margo I. Selzter. Transaction support in a log-structured file system. In *Proceedings of the Ninth International Conference on Data Engineering*, 1993.

[48] Umesh Shankar, Trent Jaeger, and Reiner Sailer. Toward automated information-flow integrity verification for security-critical applications. In *Proceedings of the ISOC Networked and Distributed Systems Security Symposium*, 2006.

[49] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the EuroSys conference*, pages 275–287, 2007.

[50] 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.

[51] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wrigh. Enabling transactional file access via lightweight kernel extensions. In *Procceedings of the 7th conference on File and storage technologies*, pages 29–42, 2009.

[52] 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.

[53] Weiqing Sun, Zhenkai Liang, R. Sekar, and V.N. Venkatakrishnan. One-way isolation: An effective approach for realizing safe execution environments. In *Proceedings of Network and Distributed System Security Symposium*, February 2005.

[54] Weiqing Sun, R. Sekar, Gaurav Poothia, and Tejas Karandikar. Practical proactive integrity preservation: A basis for malware defense. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2008.

[55] Miklos Szeredi. File system in user space (fuse). `http://fuse.sourgeforge.net`.

[56] David Wichers, Douglas Cook, Ronald Olsson, John Crossley, Paul Kerchen, Karl Levitt, and Raymond Lo. Pacl's: An access control list approach to anti-viral security. In *Proceedings of the 13th National Computer Security Conference*, pages 340–349, 1990.

[57] Wikipedia. Confused deputy problem, January 2012. `http://en.wikipedia.org/wiki/Confused_deputy_problem`.

[58] Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Amino: Extending acid semantics to the file system. *ACM Transactions on Storage (TOS)*, 3(4), June 2007.

[59] Huagang Xie. Linux intrusion detection system (lids) project. `http://www.lids.org`.

[60] 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.