# Practical Information Flow Based Techniques to Safeguard Host Integrity

A Dissertation Presented
by
Weiqing Sun

to
The Graduate School
in Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy
in
Computer Science
Stony Brook University

May 2008

**Stony Brook University**

The Graduate School

**Weiqing Sun**

We, the dissertation committee for the above candidate for
the degree of Doctor of Philosophy,
hereby recommend acceptance of this dissertation.

Professor R. Sekar, (Advisor)
Computer Science Department, Stony Brook University

Professor Erez Zadok, (Chairman)
Computer Science Department, Stony Brook University

Professor Scott Stoller, (Committee Member)
Computer Science Department, Stony Brook University

Professor Vinod Ganapathy, (External Committee Member)
Computer Science Department, Rutgers University

This dissertation is accepted by the Graduate School.

Lawrence Martin
Dean of the Graduate School

**Abstract of the Dissertation**

# Practical Information Flow Based Techniques to Safeguard Host Integrity

by

**Weiqing Sun**

**Doctor of Philosophy**

in

**Computer Science**

Stony Brook University

2008

Security threats have escalated rapidly over the past few years. Malware, zero-day attacks and rootkits are now common terms heard over the media, drawing attention from large enterprises to regular computer users. What makes it worse is that cyber crime has become financially lucrative, leading to the formation of organizations that specialize in the development and trading of malware. As a result, computer attacks have become more sophisticated and more stealthy, and can evade most of today's defenses.

Current defensive approaches like code analysis and behavior blocking can be either difficult to utilize or be evaded by indirect attacks. In contrast, techniques based on information-flow blocking can provide assurances about system integrity even in the face of sophisticated attacks. However, there has not been much success in applying information flow based techniques to modern COTS operating systems to provide satisfactory results in the aspects of security, usability, and scope. This is, in part, due to the fact that a strict application of information flow policy can break existing applications. Another important factor is the difficulty of policy development. We therefore develop two approaches in an effort to address these issues.

*SEE* (Safe Execution Environment) is suitable for running stand-alone untrusted applications in a secure way. It employs *one-way isolation:* processes running within the *SEE* are given read-access to the environment provided by the host OS, but their write operations are prevented from escaping outside the *SEE*. As a result, *SEE* processes cannot impact the behavior of host OS processes, or the integrity of data on the host OS. It provides a convenient way for users to inspect system changes made within the *SEE*. If the user does not accept these changes, they can be rolled back at the click of a button. Otherwise, the changes can be "committed" so as to become visible outside the *SEE*. We provide consistency criteria that ensure semantic consistency of the committed results. Our implementation results show that most software, including fairly complex server and client applications, can run successfully within the *SEE*. The approach introduces low performance overheads, typically below 10%.

The second approach *PPI* (Practical Proactive Integrity Preservation) aims at providing integrity guarantees at the whole system level. It focuses on proactive integrity protection by decoupling integrity labels from low-level policies that specify how to resolve accesses causing information flows that may compromise integrity. Therefore, a richer set of security levels, and more flexible policy choices can be specified to promote usability. We then develop an analysis technique that can largely automate the generation of integrity labels and policies that preserve the usability of applications in most cases. The evaluation of our implementation on Linux desktop distributions indicates that it can stop a variety of sophisticated malware attacks, while remaining usable.

To my wife Yi,

my daughter Haibei,

and my parents.

# Contents

**List of Tables** **x**

**List of Figures** **xi**

**Acknowledgments** **xii**

**1 Introduction** **1**

1.1 Current Techniques in System Integrity Protection . . . . . . 2

1.2 Mechanisms in Restricting Information Flow and Our Approaches 3

1.3 Dissertation Organization . . . . . . . . . . . . . . . . . . . . . 7

**I Safe Execution Environments** **9**

**2 Overview of *SEE* Approach** **10**

2.1 Motivating Applications . . . . . . . . . . . . . . . . . . . . . . 10

2.2 *SEE* Requirements and the Need for New Approach . . . . . . 11

2.3 Approach Overview . . . . . . . . . . . . . . . . . . . . . . . . . 13

2.3.1 Achieving One-way Isolation . . . . . . . . . . . . . . 14

2.3.2 Committing Changes . . . . . . . . . . . . . . . . . . . 17

**3 Design and Implementation of *SEE*** **19**

3.1 Isolation File System (IFS) . . . . . . . . . . . . . . . . . . . . 19

3.1.1 High-Level Overview . . . . . . . . . . . . . . . . . . . 19

3.1.2 Design Details . . . . . . . . . . . . . . . . . . . . . . . 20

3.2 Implementation of IFS Commit Operation . . . . . . . . . . . 23

3.2.1 Commit Criteria . . . . . . . . . . . . . . . . . . . . . . 25
3.2.2 Efficient Implementation of Commit . . . . . . . . . . . 30
3.3 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
3.3.1 Implementing Restriction at System Call Layer. . . . . 35
3.3.2 Support for Network Operations. . . . . . . . . . . . . 37
3.3.3 User Interface. . . . . . . . . . . . . . . . . . . . . . . . 38

**4 Evaluation of *SEE*** **40**
4.1 Evaluation of Functionality . . . . . . . . . . . . . . . . . . 40
4.2 Evaluation of Performance . . . . . . . . . . . . . . . . . . . 43

**5 Related Work of *SEE*** **45**

**II Practical Proactive Integrity Preservation** **51**

**6 Overview of *PPI* Approach** **52**
6.1 Goals of Approach . . . . . . . . . . . . . . . . . . . . . . . 55
6.2 Salient Features . . . . . . . . . . . . . . . . . . . . . . . . . 56

**7 Design and Implementation of *PPI*** **58**
7.1 Policy Framework . . . . . . . . . . . . . . . . . . . . . . . . 58
7.1.1 Trust and Integrity Levels . . . . . . . . . . . . . . . . 58
7.1.2 Integrity Labels Versus Policies . . . . . . . . . . . . . 60
7.2 Automating Policy Development . . . . . . . . . . . . . . . . 62
7.2.1 Computing Dependencies, Contexts and Labels . . . . 63
7.2.2 Policy Generation . . . . . . . . . . . . . . . . . . . . . 65
7.2.3 Soundness of Policies . . . . . . . . . . . . . . . . . . . 69
7.3 Limiting Trust . . . . . . . . . . . . . . . . . . . . . . . . . . 70
7.3.1 Limiting Trust on Key Applications . . . . . . . . . . . 72
7.4 Enforcement Framework . . . . . . . . . . . . . . . . . . . . 73

**8 Evaluation of *PPI*** **78**
8.1 Experience . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78

8.2 Effectiveness against Malicious Applications . . . . . . . . . . . 81
8.3 Usability of *PPI* Versus Low Watermark . . . . . . . . . . . . 82
8.4 Performance Overheads . . . . . . . . . . . . . . . . . . . . . . . . . 84
8.5 Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84

**9 Related Work of *PPI*** **86**

**10 Conclusion** **90**

# List of Tables

1 Comparison for Log-based Commit and State-based Commit. All numbers are in seconds. . . . . . . . . . . . . . . . . . . . . . 44
2 *PPI* Policy Generation in Different Phases. . . . . . . . . . . . 80
3 Microbechmark Result using LMbench. All numbers in microseconds. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
4 Application Performance Overhead. All numbers are in seconds averaged across 10 runs. . . . . . . . . . . . . . . . . . . . . . . 84

# List of Figures

1 Architecture of *SEE* . . . . . . . . . . . . . . . . . . . . . . . . 15

2 Illustration of IFS Layout on Modification Operations . . . . . 21

3 Performance Results for Program Execution in *SEE* . . . . . . 42

4 Classification of Applications in *PPI* Policy Framework . . . . 59

5 *PPI* System Architecture . . . . . . . . . . . . . . . . . . . . . . 74

# Acknowledgments

I want to express my deepest gratitude to my advisor, Prof. R. Sekar, for his valuable advising, and invaluable support through my Ph.D. program. During this long journey, his guidance and encouragement have helped me to keep focused on my research. I have been greatly benefited from his dedication and high standard on research.

I would like to thank my dissertation committee members Prof. Erez Zadok, Prof. Scott Stoller, and Prof. Vinod Ganapathy for their insightful comments and suggestions.

Secure Systems Lab has been a wonderful workplace. It provides powerful computing facilities, spacious cubicles, a good collection of technical books and comfortable sofas. And most importantly, I have enjoyed working with outstanding fellow lab members. I am especially grateful to V.N. Venkatakrishnan, Zhenkai Liang, Wei Xu and Prem Uppuluri for their direct support at different stages of my research work. I want to thank Alok Tongaonkar, Sandeep Bhatkar, Lorenzo Cavallaro and Yves Younan for their immense help on my writing and presentation. I have enjoyed working with Krishna Kumar, Varun Katta, Munyaradzi Chiwara, Gaurav Poothia, Aditya Rekha Dharanipragada, Milan Manavat, Manish Nair, Tejas Karandikar, Kumar Thangavelu, Anupama Chandwani, Srivani Narra, Ravi Kiran, Abhishek Dhamija, Sreenath Vasudevan and Amol Karmarkar. I would like to mention other lab members with whom I have had enjoyable experience: Ajay Gupta, Tianning Li, Prateek Saxena, Abhishek Chaturvedi, Vishwas Nagaraja, Shruthi Murthy, Yogesh Chauhan, Divya Padbhanabhan, Mohan Channa, Tapan Kamat, Sandhya Menon, Karthik Sreenivasa Murthy, Mithun Iyer, Xiaojun Wang, Jun Yuan and several others.

I would like to thank my other friends in Stony Brook, in particular, Wei Zhang, Chen Ling, Hui Zhang, Zongheng Zhou, Jie Chen, Peiyan Wang, Yang Wang, Rahul Agarwal, Dimitris Papamichail, Yiguo Wu, for their friendship and help.

Finally, my special thanks are to my family: my parents, my sister and my in-laws for their continued support; my lovely 9-month-old daughter Haibei for bringing immense happiness in my life; and most importantly my wife Yi for her love, understanding, and care.

# CHAPTER 1

# Introduction

"We are fielding 2,000 attacks per hour", said Microsoft. "We see as many as 60,000 come on in a day", said Symantec on the number of infected systems with bot software. "My computer got a virus after I clicked an email, what should I do?", complained by an anonymous computer user, ......

Security threats have escalated rapidly over the past few years. Malware, zero-day attacks and rootkits are now common terms heard over the media, drawing attention from large enterprises to regular computer users. What makes it worse is that cyber crime has become financially lucrative, leading to the formation of organizations that specialize in the development and trading of malware. As a result, computer attacks have become more sophisticated and more stealthy, and can evade most of today's defenses.

As an indicator for the trustworthiness of the system, system integrity is a fundamental concern in security. It is usually first violated as a result of a successful attack. Even attacks that focus on stealing user information need to first break system integrity in order to plant "agent" software, e.g., spyware. Hence, in this dissertation, we focus on developing solutions that can safeguard system integrity from malware attacks.

## 1.1 Current Techniques in System Integrity Protection

Current defensive techniques in system integrity protection against untrusted code[1] can be broadly divided into the following categories:

- *Code analysis* focuses on identifying security violations by performing analysis on the code. Code analysis can be static or dynamic. Static code analysis involves analyzing the code (usually in binary format) of the untrusted application without actually executing it. However it is a difficult task by its own nature. Dynamic code analysis observes its behavior by actually running the code with possible test data sets. But more and more malware incorporate anti-analysis mechanisms to thwart these analyses. Therefore, code analysis is not a practical approach against new types of adaptive malware.

- *Behavior blocking* involves monitoring of untrusted code at runtime and blocking behaviors that are deemed malicious. Sandboxing is one representative in this category, and it is typically achieved by restricting the set of resources (such as files) that can be written by untrusted processes. The main drawback of this technique is the difficulty of policy selection, that is, it is not easy to determine what actions are permissible for a given application: restrictive policies are likely to break normal execution of benign applications, while permissive policies lead to security holes. Moreover, attackers can employ indirect attacks, which involve multiple steps that gradually gain unauthorized privileges. For instance, an attacker may first modify `.bashrc`[2] to create an alias for the command `sudo`[3], and then waits for the user to run `sudo`, when the

[1]We use the term "untrusted software" to refer to software obtained from untrusted sources on the Internet. Untrusted software may be malicious or non-malicious. On the other hand, benign software, which is obtained from trusted sources, is assumed to be non-malicious.

[2].bashrc is the user-specific Bash init file in which user can define functions and alias for various bash commands.

[3]sudo allows a user to execute a command as the superuser or another user.

attacker's trojan program will be invoked. Since it is hard to identify all the possible intermediate files, sophisticated and stealthy attacks can evade sandbox-like techniques.

With indirect attacks, an attacker seeks to influence some benign applications, possibly through multiple steps. We observe that *information-flow blocking* techniques can effectively defeat such attacks by blocking information flows from untrusted to benign applications. Policies that restrict information flows are more natural for providing assurances about end-to-end system integrity.

## 1.2 Mechanisms in Restricting Information Flow and Our Approaches

In a simplified model, where benign and untrusted subjects/objects[4] co-exist on the system, unacceptable information flow can happen as a result of (a) a benign subject reading an untrusted object, or (b) an untrusted subject writing to a benign object or communicating with a benign subject. Given such a model, there are the following possible ways to restrict information flow.

### Isolation Approaches.

Isolation is a straightforward way to achieve restriction of information flows. An untrusted subject can be isolated so that it can no longer affect benign subjects or objects outside the isolated environment. Usually, in order for an isolated subject to exhibit the same behavior as it runs natively on the host environment, all the objects it will access need to be duplicated into an isolated environment. An isolation-based approach is especially suitable for executing stand-alone untrusted applications. For instance, suppose that a user downloads a photo album utility from an untrusted website, and wants to try it on

[4]Subjects refer to processes, while objects refer to files, sockets, etc.

her own photo files. In this case, isolation would come in handy to prevent information flows from this untrusted utility to benign subjects/objects.

Current virtual machine based approaches [67] employ two-way isolation between a host and guest operating system. The "playground" approaches developed for Java programs in [36, 12] also belong to this general category. However, two-way isolation is too heavy-weight because of the extra burden of environment duplication. In the above album utility example, the user has to manually copy all her photo files into the isolated environment before using the utility to process them. It also has the inconvenience of requiring an explicit copy operation to make the results of untrusted execution visible in the host system.

To mitigate these drawbacks, we propose a new technique called *Safe Execution Environments* (*SEE*), as elaborated in Part I of this dissertation. This approach uses a *one-way isolation* technique. It makes the host state visible inside the *SEE* so that accurate environment reproduction is assured. At the same time, it prevents information flow from untrusted applications to benign applications, so benign applications cannot see the file objects created or modified by untrusted applications, and hence cannot be compromised by untrusted applications. To achieve this, we *redirect* any modification operation made within the *SEE* to a different resource that is invisible outside the *SEE*. If the result of an *SEE* execution is deemed safe, our *SEE* provides a "commit" functionality to make the results of untrusted execution visible to the host system.

## Approaches Based on Runtime Enforcement.

Although isolation based technique is effective in restricting information flows without affecting the usability of untrusted applications, there is one problem it cannot solve by itself: users need to decide whether the results of untrusted execution are "safe" to be committed to the host system. For instance, in the above photo album example, the user has to decide whether to commit the modified photo files back to the host system. It turns out not to be an easy task because users usually lack the necessary knowledge in how the modified files are going to be used afterwards. An alternative approach that overcomes this drawback is that of labeling the outputs of untrusted execution so that additional precautions can be taken at the point where these results are actually used by benign subjects. This line of thought directs us to information flow approaches based on runtime enforcement.

Unlike the isolation approaches, runtime enforcement based approaches provide comprehensive runtime information flow tracking and policy enforcement. *Biba* [8], as a basic model, has a strict “no read down and no write up” policy, which prevents any information flowing from low to higher integrity level. However, the past 30+ years effort in applying this model to real-world systems has not been much of a success. The task of labeling all the subjects/objects on the system seems daunting. Moreover, the strict policy causes usability problems, preventing some benign applications from being used successfully. For example, a file copying utility should be usable on high as well as low integrity files, with the copy inheriting the integrity of the original. Hence, there are the following efforts aimed at producing more usable systems by relaxing this basic model.

- *No write up, but read down permitted.* Windows Vista is one example in this category, which divides the set of resources in the system into several integrity levels, and permits a process to overwrite a file only if its integrity level dominates that of the file. But techniques that regulate write-access without restricting read-access are not sufficient to thwart adaptive attacks such as indirect attacks, where a benign application ends up consuming malicious outputs stored in the file system. For instance, user-specific customization files and scripts can lead to possible indirect attacks as we mentioned before, but it is difficult to identify all such files before hand. Worse, a benign application may be compromised by an untrusted input file. Clearly, most of the possible inputs to an application cannot be predicted in advance.

- *No read down, but write up allowed. Back to the Future* system [21] enforces the “no read down” policy, but not the “no write up” policy. Untrusted applications can freely write to the file system. But it can recognize any attempt by malware to inject itself into inputs consumed by benign applications. The disadvantage of this approach is that of delayed detection: malware actions are not stopped at the point where they overwrite critical files, but at the point where a benign application uses them. This inconveniences the user, as her attempts to run that application would fail.

- *Low Watermark.* This model introduces the “subject downgrade” policy over Biba model, so that a subject can be downgraded to lower level after reading a low level object. This addresses some of the usability issues, such as problems associated with running the above-mentioned copy utility, but as described in [19], it now suffers from the “self-revocation” problem.

Given that the existing relaxed models are not satisfactory, we propose our second approach, *Practical Proactive Integrity protection* (*PPI*) in an effort to address the above drawbacks. This technique, described in Part II of this dissertation, follows the direction of traditional runtime enforcement based integrity protection models with important and innovative enhancements. Specifically,

- *Flexible decomposition of high-level policies into low-level policies.* In traditional approaches, labels effectively define the access policies: a subject is permitted to read (write) an object only if the subject’s integrity is equal to or lower (equal to or higher) than that of the object. In contrast, we distinguish between *labels,* which are a judgment of the trustworthiness of an object (or subject), from *policies* that state whether a certain read or write access should be permitted. Based on this separation, our approach allows integrity levels of objects or subjects to change over their lifetime. Moreover, “read-down” and “write-up” conflicts are resolved differently for different objects and subjects. These factors provide flexibility in developing low-level policies that preserve system integrity without unduly impacting usability.

- *Automated analysis for generating enforceable policies.* Given the large number of objects (hundreds of thousands) and subjects (thousands), manual configuration of policies for every object/subject pair is impractical. We therefore develop techniques that utilize an analysis of access patterns observed on an unprotected system to automatically derive policies. This analysis can also be used to automatically complete the set of integrity-critical applications, starting from a partial list provided by a policy developer.

- *A flexible enforcement framework.* Our enforcement framework, consists of a small, security-critical enforcement component that resides in the OS kernel, and a user-level component that incorporates more complex features that enhance functionality without impacting security. This framework also incorporates features needed for learning and synthesizing policies for new applications.

- *Mechanisms for limiting trust.* There are some instances when high-integrity applications should be allowed to access low-integrity files. We develop techniques that enable such exceptions to be restricted. Our techniques typically have the effect of distinguishing between code/configuration inputs from data inputs, and ensuring that exceptions are made only for data inputs. Using these mechanisms, we describe how we can limit the amount of trust placed on important applications such as software installers, web browsers and email handlers, and file utilities.

## 1.3 Dissertation Organization

The rest of the dissertation is organized as follows. Part I describes our first technique. In particular, Chapter 2 presents an overview of the *SEE* approach. Chapter 3 describes the design and implementation of *SEE*. Chapter 4 provides an evaluation of the functionality as well as the performance of the *SEE* approach. Related work of *SEE* is discussed in Chapter 5. Part II describes our second technique. We present an overview of the *PPI* approach in Chapter 6. The design and implementation of the *PPI* approach is described in Chapter 7, followed by an evaluation of its functionality and performance in Chapter 8. Related work for *PPI* is discussed in Chapter 9. Finally, Chapter 10 concludes this dissertation.

# Part I

# Safe Execution Environments

# CHAPTER 2

# Overview of *SEE* Approach

## 2.1 Motivating Applications

System administrators and desktop users often encounter situations where they need to experiment with potentially unsafe software or system changes. A high-fidelity *safe execution environment (SEE)* that can support these activities, while protecting the system from potentially harmful effects, will be of significant value to these users. Applications of such *SEE* include:

- *Running untrusted software.* Often, users execute downloaded freeware, shareware or mobile code. The risk of damage to the user's computer system due to untrusted code is high, yet a significant fraction of users seem to be willing to take this risk in order to benefit from the functionality offered by such code. An *SEE* can minimize security risks without negating the functionality benefits provided by such software.

- *Vulnerability testing.* System administrators may be interested in probing whether a system is susceptible to the latest email virus, worm or other attacks. A high-fidelity *SEE* can allow them to perform such testing without the risk of compromising production systems.

- *Software updates/patches.* Application of security patches are routinely delayed in large enterprises in order to allow time for compatibility and interoperability testing. Such testing is typically done after shutting down production systems for extended periods, and hence may be scheduled for weekends and holidays. In contrast, a high-fidelity *SEE* can allow testing of updates to be performed without having to shutdown production systems. These concerns apply more generally to software upgrades or installations as well.

- *System reconfiguration.* Administrators may need to reconfigure software systems, and would ideally like to "test out" these changes before deploying them on production systems. This is currently accomplished manually, by saving backup copies of all files that may be modified during reconfiguration. An *SEE* will automate this process, and moreover, avoid pitfalls such as overlooking to backup some of the modified files.

## 2.2 *SEE* Requirements and the Need for New Approach

In order to support the kinds of applications mentioned above, an *SEE* must provide the following features:

- *Confinement without undue restrictions on functionality.* The effects of process execution within an *SEE* should not "escape" the *SEE* and become visible to normal applications running outside. Otherwise, one cannot rule out the possibility of *SEE* processes altering the operation of other applications running on the same system or elsewhere in the network. Such confinement can be achieved using access control restrictions, e.g., by prohibiting all operations that modify files or access the network. However, such restrictions will prevent most applications from executing successfully within an *SEE*.

- *Accurate environment reproduction.* For *SEEs* to be useful in the above applications, it is essential that the behavior of applications be identical, whether or not they operate within the *SEE*. Since the behavior of an application is determined by its environment (contents of configuration or data files, executables, libraries, etc.), it is necessary to reproduce, as accurately as possible, the same environment within the *SEE* as the environment that exists outside *SEE*.

- *Ability to commit results.* In many of the above applications, including untrusted software execution and software or system updates, a user would like to retain the results of activities that were successful. Thus, the *SEE* must provide a mechanism to "commit" the results of activities that took place within it. A successful commit should have the same effect as if all of the operations carried out within the *SEE* actually took place outside.

Most existing approaches for safe execution do not satisfy these requirements. For instance, sandboxing techniques achieve confinement, but do so by severely restricting functionality. Virtual machines (VMs) and related approaches [11, 71] relax access restrictions, but do not offer any support for environment reproduction or committing. File versioning systems [54, 80, 79, 39, 13, 50, 52, 62, 44] can provide rollback capabilities, but they don't provide a mechanism to discriminate among changes made by different processes, and hence cannot support selective rollback of the effects of untrusted process execution.

The concept of *isolation* has been proposed as a way to address the problem of effect containment for compromised processes in [24, 33, 57]. [33] developed the concept of *one-way isolation* as an effective means to isolate the effects of running processes from the point they are compromised (or suspected of being compromised). They also develop protocols for realizing one-way isolation in the context of databases and file systems. However, they only provide a high-level treatment, and do not address practical issues that arise in implementing such an approach for COTS applications running over commodity OSes.

[32] addressed some of these issues and developed a user-level tool for isolating the effects of COTS applications on the Linux OS. The focus of that effort was on untrusted software execution, and on a solution that was

realized entirely at the user level. Such a solution does not require OS changes or even administrative privilege to install or use the tool. However, in order to achieve a completely user-land solution, [32] compromises on performance as well as generality. In particular, the approach suffers from high overheads that can be over 100% in some instances. Moreover, isolation semantics cannot be faithfully reproduced for operations that concern file meta-data such as permissions and ownership. For directories, isolation is achieved using an ad-hoc approach that is hard to implement and provides semantics that is inconsistent with that of files. Finally, no systematic solution to the commit problem is provided. The approach developed in this part addresses all these drawbacks. Moreover, it generalizes the approach so that isolation can be provided for non-file operations, e.g., certain classes of network accesses.

## 2.3 Approach Overview

The *SEEs* are based on the concept of one-way isolation. Whereas VMs generally employ two-way isolation between the host environment and the environment that exists within a VM, one-way isolation makes the host environment visible within the *SEE*. In this sense, the *SEE* processes can (and do) see the environment of their host system, and hence accurate reproduction of environment is assured. However, the effects of *SEE* processes are isolated from outside applications, thereby satisfying the confinement requirement.

In our approach, an *SEE* is created to run a process whose effects are to be shielded from the rest of the system. One or more such *SEEs* may be active on the host OS. Any children created by processes within an *SEE* will also be confined to that *SEE*, and will share the same consistent view of system state. Typically, a user will start a command shell within a new *SEE*, and use this shell to carry out tasks such as running untrusted programs. She may also run helper applications, such as image or document viewers, or arbitrary utility applications to examine the resulting system state. Finally, if she wants to accept the changes made within the *SEE*, she can commit the results. The commit process causes the system state, as viewed inside the *SEE*, to be merged with the state of the host OS. We present consistency criteria aimed at ensuring the correctness of the results of the commit process.

Our approach is implemented using interposition at the system call and virtual file system layers, and hence does not require any changes to applications that run inside the *SEE*. Even complex tasks such as compilation and installation of large software packages, and execution of complex client and server applications can be carried out successfully within the *SEE*. This is because our approach places few restrictions on operations performed by most applications. In particular, no restrictions are placed on file accesses, except in the case of access to special devices. Network operations that correspond to "read" operations, such as querying a name server, can be permitted as well. Network accesses that correspond to "write" operations can be permitted when the target of the communication satisfies one of the following conditions:

- it is an application running within an *SEE*, possibly on a different host, or
- it is a special-purpose "proxy" that is layered between the application and the service accessed by it, and can buffer the write actions until commit time.

The key challenge in implementing such proxies is that even though they buffer certain operations, they should provide a consistent view of system state to the *SEE* applications. Specifically, if an *SEE* process "writes" to such a proxy and subsequently performs a "read" operation, the proxy should return the result that would have been returned if the write operation had actually been carried out.

### 2.3.1 Achieving One-way Isolation

Figure 1 illustrates the overview of our *SEE*. It is a layer between the isolated program and the operating system, which is based on intercepting and manipulating the requests made by the isolated program. The primary goal of isolation in our approach is effect containment: preventing the effects of

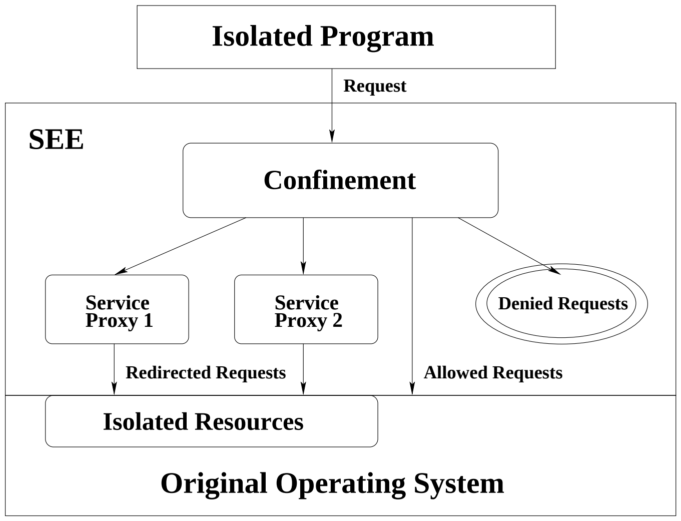


Figure 1: Architecture of *SEE*

*SEE* processes from affecting the operation (or outcome) of processes executing outside the *SEE* [1]. This means that any "read" operation (i.e., one that queries the system state but does not modify it) may be performed by *SEE* processes. It also means that "write" operations should not be permitted to affect system state. There are two options in this context: one is to *restrict* the operation, i.e., disallow its execution. The second option is to *redirect* the operation to a different resource that is invisible outside the *SEE*. Once a write operation is redirected, it is important that subsequent read operations on the same resource be redirected as well.

By *restriction,* we mean that an operation is prevented from execution. An error code may be returned to the process, or the operation may be silently suppressed and a success code returned. In either case, restriction is easy to implement — we need only know the set of operations that can potentially alter system state. The main drawback of restriction is that it will likely prevent applications from executing successfully. For instance, if a program writes a

[1]Note that we are interested in confinement [30] from the point of view of system integrity, rather than confidentiality. As such, we do not deal with with issues such as covert channels.

file, it expects to get back the same content at a later point in the program when the file is read. However, an approach based on restriction cannot do this, and hence most nontrivial applications will fail to run successfully under such restriction. For this reason, restriction is a choice of last resort in our approach.

By *redirection,* we mean that any operation that modifies some component of the host environment is instead redirected to a different component that is not accessed by the host OS processes. For instance, when an *SEE* process tries to modify a file, a copy of the original file may be created in a "private" area of the file system, and the modification operation redirected to this copy. Redirection is intended to provide a consistent view of system state to a process, thereby allowing it to run successfully.

Redirection can be *static* or *dynamic.* Static redirection requires the source and target objects to be specified manually. It is ideal for network operations. For instance, one may statically specify that operations to bind a socket to a port $p$ should be redirected to an alternate port $p'$. Similarly, one may specify that operations to connect to a port $p$ on host $h$ should be redirected to host $h'$ (which may be the same as $h$) and port $p'$. By using such redirection, we can build *distributed SEEs*, where processes executing within *SEEs* on multiple hosts can communicate with each other. Such distributed *SEEs* are particularly useful for safe execution of a network server application, whose testing would typically require accesses by nonlocal client applications. (Note, however, that this approach for distributed *SEEs* works only when all cross-*SEE* communications take place directly between the *SEE* processes, and not through other means, e.g., indirect communication through a shared NFS directory.)

Static redirection becomes infeasible if the number of possible targets is too large to be enumerated in advance. For instance, it is hard to predict the files that may be accessed by an arbitrary application. Moreover, there are dependencies among operations on different file objects, e.g., an operation to create a file has the indirect effect of changing the contents of the directory in which the file is created. Simply redirecting an access on the file, without correspondingly modifying accesses of the directory, won't work. To handle such complexities, our approach supports *dynamic redirection,* where the target for redirection is determined automatically during the execution of *SEE* processes. However, the possibility of hidden dependencies means that the implementation of dynamic redirection may have to be different for different kinds of objects. Specifically, in our *SEE* architecture, (Figure 1), dynamic redirection is supported by *service-specific proxies.* Currently, there is a proxy for file service, and we envision proxies for other services such as WWW or email.

In our current implementation, system call interposition is used to implement restriction and static redirection. We restrict all modification operations other than those that involve the file system and the network. In the case of file operations, all accesses to normal files are permitted, but accesses to raw devices and special purpose operations such as mounting file systems are disallowed. In terms of network operations, we permit any network access for which static redirection has been set up. In addition, accesses to the name server and X-server are permitted. (In reality, *SEE* processes should not get unrestricted access to X-server, Our current implementation solves this problem by statically redirecting X requests to a separate X-server.)

Dynamic redirection is currently supported in our implementation for only file system accesses. It is realized using a proxy called the Isolation File System (IFS), which is described in detail in Section 3.1.

### 2.3.2 Committing Changes

There are two key challenges in committing: one is to ensure *consistency* of the resulting system state; the other is *efficiency* — to reduce the space and time overheads for logging and re-running of operations to a level that provides good performance. Below, we provide a high-level overview of the issues involved in commit.

The key problem in terms of consistency is that a resource accessed within the *SEE* may have been independently accessed outside of the *SEE*. This corresponds to concurrent access on the same resource by multiple processes, some within *SEE* and some outside. One possible consistency criterion is the serializability criterion used in databases. Other consistency criteria may be appropriate as well, e.g., for some text files, it may be acceptable to merge the changes made within the *SEE* with changes made outside, as long as the changes involve disjoint portions of the file. A detailed discussion of the issues involved in defining commit criteria is presented in Section 3.2.1.

There may be instances where the commit criteria may not be satisfied. In this context, we make the following observations:

- There is no way to guarantee that results can be committed automatically and produce consistent system state, unless we are willing to delay or disallow execution of some applications on the host OS. Introducing restrictions or delays on host OS processes will defeat the purpose of *SEE*, which is to shield the host OS from the actions of *SEE* processes. Hence this option is not considered in our approach.

- If the results are not committed, then the system state is unchanged by tasks carried out within the *SEE*. This means that these tasks can be rerun, and will most likely have the same desired effect. Hopefully, the conflicts were the results of infrequent activities on the host OS, and won't be repeated this time, thus enabling the results to be committed.

- If retry isn't an option, the user can manually resolve conflicts, deciding how the files involved in the conflict should be merged. In this case, the commit criteria identifies the files and operations where manual conflict resolution is necessary.

As a final point, we note that if a process within an *SEE* communicated with another process executing within a different *SEE*, then all such communicating *SEEs* need to be committed as if they were part of a single distributed transaction. Currently, our implementation does not support distributed commits. Our approach for committing the results of operations performed within a single *SEE* is described in Section 3.2.

# CHAPTER 3

# Design and Implementation of *SEE*

## 3.1 Isolation File System (IFS)

### 3.1.1 High-Level Overview

In principle, a file system can be viewed as a tree structure. Internal nodes in this tree correspond to directories or files, whereas the leaves correspond to disk blocks holding file data. The children of directory nodes may themselves be directories or files. The children of file nodes will be disk blocks that either contain file data, or pointers to file data.

This view of file system as a tree suggests an intuitive way to realize one-way isolation semantics for an entire file system: when a node in the original file system is about to be modified, a copy of this node, as well as all its ancestors, is created in a "private" area of the file system called *temporary storage*. The write operation, as well as all other subsequent operations on this node, are then redirected to this copy.

In essence, we are realizing isolation using copy-on-write. Although the copy-on-write technique has been used extensively in the context of plain files, it has not been studied in the context of directories. Realizing IFS requires us to support copy-on-write for the entire file system, including directories and plain files.

In our approach, copy-on-write on directories is supported using a *shallow-copy* operation, i.e., the directory itself is copied, but its entries continue to point to objects in the original file system. In principle, one can use shallow-copy on files as well, thus avoiding the overhead of copying disk blocks that may not be changed within the IFS. However, the internal organization of files is specific to particular file system implementations, whereas we want to make IFS to be file-system independent. Hence files are copied in their entirety.

IFS is implemented by interposing file system operations within the OS kernel at the Virtual File System (VFS) layer. VFS is a common abstraction in Unix across different file systems, and every file system request goes through this layer. Hence extensions to functionality provided at VFS layer can be applied uniformly and transparently to all underlying file systems such as `ext2`, `ext3` and NFS.

We realize VFS layer interposition using the stackable file system approach described in [76]. In effect, this approach allows one to realize a new file system that is "layered" over existing file systems. Accesses to the new file system are first directed to this top layer, which then invokes the VFS operations provided by the lower layer. In this way, the new file system extends the functionality of existing file systems without the need to deal with file-system-specific details.

### 3.1.2 Design Details

The description in the previous section presented a simplified view of the file system, where the file system has a tree-structure and consists of only plain files and directories. In reality, UNIX file systems have a DAG (directed acyclic graph) structure due to the presence of hardlinks. In addition, file systems contain other types of objects, including symbolic links and special device files. As mentioned earlier, IFS does not support special device files. An exception to this rule is made for `pty`'s and `tty`'s, as well as pseudo devices like `/dev/zero`, `/dev/null`, etc. In these cases, access is redirected to the

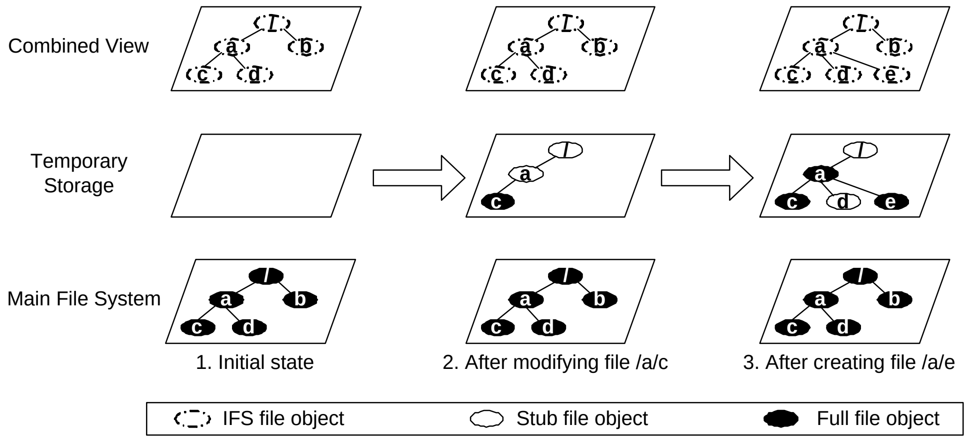


Figure 2: Illustration of IFS Layout on Modification Operations

corresponding device files on the main file system. A symbolic link is simply a plain file, except that the content of the file is interpreted as the path name of another file system object. For this reason, they don't need any special treatment. Thus, we need only describe how IFS deals with hard links (and the DAG structure that can result due to their use.)

When the file system is viewed as a DAG, its internal nodes correspond to directories, and the leaves correspond to files. As mentioned earlier, the IFS does not look into the internal structure of files, and hence we treat them as leaf objects in the DAG. All nodes in the DAG are identified by a unique identifier called the *Inode number*. (The inode number remains unique across deletion and recreation of file objects.) The edges in the DAG are *links*, each of which is identified by a name and the Inode number of the object pointed by the link. This distinction between nodes and links in the file system plays a critical role in every aspect of IFS design and implementation.

Figure 2 illustrates the operation of IFS. The bottom layer corresponds to a host OS file system. The middle layer is the temporary storage to hold modified copies of files and directories. The top layer shows the view within IFS, which is a combination of the views in the bottom two layers. Note that the ordering of the bottom two layers in the figure is significant: the view contained in the temporary storage overrides the view provided by the main file system.

The temporary storage area is also known as "private storage area" to signify that fact that it is not to be accessed by the host OS. In order to support efficient movement of files between the two layers, which is necessary to implement the commit operation efficiently, it is preferable that the temporary storage be located on the same file system as the bottom layer. (If this is not possible, then temporary storage can be on a different file system, with the caveat that committing will require file copy operations as opposed to renames.) Henceforth, we will use the term *main file system* to denote the bottom layer and *IFS-temporary storage* (or simply "temporary storage") to refer to the middle layer.

In addition to storing private copies of files modified within the *SEE* in the temporary storage, the IFS layer also contains a table that maintains additional information necessary to correctly support IFS operation. This table, which we call as *inode table*, is indexed by the inode numbers of file system objects. It has a field indicating that whether the inode corresponds an object in temporary storage (temp) or an object the main file system (main). Further, if it is an object in the temporary storage, the flag indicates whether it is a stub object (stub). A stub object is simply a reference to the version of the same object stored in the main file system. In addition, auxiliary information needed for the commit operation is also present, as described in Section 3.2.

In our IFS implementation, copy-on-write of regular files is implemented using normal file copy operations. In particular, when a plain file $f$ is modified for the first time within the *SEE*, a stub version of all its ancestor directories is created in temporary storage (if they are not already there). Then the file $f$ is copied into temporary storage. From this point on, all references to the original file will be redirected to this copy in temporary storage.

After creating a copy of $f$, we create an entry in the inode table corresponding to the original version of $f$ on the main file system. This is done so as to handle hard links correctly. In particular, consider a situation when there is a second hard link to the same file object, and this link has not yet been accessed within IFS. When this link is subsequently accessed, it will be referencing a file in the main file system. It is necessary to redirect this reference to the copy of $f$ in temporary storage, or otherwise, the two links within IFS that originally referred to the same file object will now refer to different objects, thereby leading to inconsistencies.

The copy-on-write operation on directories is implemented in a manner similar to that of files. Specifically, a stub version of the directory's ancestor nodes are first created in temporary storage. Next, the directory itself is copied. This copy operation is a *shallow copy* operation, in that only a stub version of the objects listed in the directory are created.

We illustrate the operation of IFS using the example shown in Figure 2. Suppose that initially (i.e., step 1 in this figure), there is a directory `a` and a file `b` under the root directory in the main file system, with files `c` and `d` within directory `a`. Step 2 of this figure illustrates the result of modifying the file `/a/c` within the *SEE*. The copy-on-write operation on `/a/c` first creates a stub version of the ancestor directories, namely, `/` and `/a`. Then the file `/a/c` is copied from the main file system to the temporary storage. Subsequent accesses are redirected to this copy in temporary storage.

The third step of Figure 2 shows the result of an operation that creates a file `/a/e` within the *SEE*. Since this changes the directory `a` by adding another file to it, a shallow copy of the directory is made. Next, the file `e` is created within the directory. The combined view of IFS reflects all these changes: accesses to file `/a/c` and `/a/e` are redirected to the corresponding copies in the temporary storage, while accesses to file `/a/d` will still go to the version in the main file system.

## 3.2 Implementation of IFS Commit Operation

At the end of *SEE* execution, the user may decide either to discard the results or commit them. In the former case, the contents of IFS are destroyed, which means that we simply delete the contents of temporary storage and leave the contents of the main file system "as is." In the latter case, the contents of the temporary storage need to be “merged” into the main file system.

When merging the contents of temporary storage and main file systems, note that conflicting changes may have taken place within and outside the IFS, e.g., the same file may have been modified in different ways within and outside the *SEE*. In such cases, it is unclear what the desired merge result should be. Thus, the first problem to be addressed in implementing the commit operation is that of identifying *commit criteria* that ensure that the commit operation can be performed fully automatically (i.e., without any user input) and is guaranteed to produce meaningful results. We describe possible commit criteria in Section 3.2.1. Following this, we describe an efficient algorithm for committing results in Section 3.2.2.

If the commit criteria is not satisfied, then manual reconciliation of conflicting actions that took place inside the *SEE* and outside will be needed. The commit criteria will also identify the set of conflicting files and operations. At this point, the user can decide to:

- *abort,* i.e., discard the results of *SEE* execution. This course of action would make sense if the activities performed inside *SEE* are no longer relevant (or useful) in the context of changes to the main file system.

- *retry*, i.e., discard the results of *SEE* execution, create a new *SEE* environment, redo the actions that were just performed within the *SEE*, and then try to commit again. If the conflict were due to activities on the host OS that are relatively infrequent, e.g., the result of a cron job or actions of other users that are unlikely to be repeated, then the retry has a high probability of allowing a successful commit. (Note that the retry will likely start with the same system state as the first time and hence will have the same net effect as the first time.)

- *resolve conflicts*, i.e., the user manually examines the files involved in the conflict (and their contents) and determines if it is safe to commit; and if so, what the merged contents of the files involved in the conflict. The commit criteria will identify the list of files involved in the conflict and the associated operations, but the rest of the steps need to be performed manually.

### 3.2.1 Commit Criteria

The commit criteria is a set of rules which determine whether the results of changes made within an *SEE* can be committed automatically, and lead to a consistent file system state. Since the problem of consistency and committing has been studied extensively in the context of database transactions, it is useful to formulate the commit problem here in the terms used in databases. However, note that there is no well-defined notion of transactions in the context of IFS. We therefore identify the entire set of actions that took place within *SEE* in isolation as a transaction $T_i$ and the entire set of actions that took place outside of the *SEE* (but limited to the actions that took place during the lifetime of the *SEE*) as another transaction $T_h$.

There are several natural choices for commit criteria:

- *Noninterference.* This requires that the actions contained in $T_i$ be unaffected by the changes made in $T_h$ and vice-versa. More formally, let $RS(T)$ and $WS(T)$ denote respectively the set of all filesystem objects read and written by a transaction $T$, respectively. Then, noninterference requires that

$$RS(T_i) \cap WS(T_h) = \phi$$

$$RS(T_h) \cap WS(T_i) = \phi$$

$$WS(T_i) \cap WS(T_h) = \phi$$

  The advantage of this criteria is that it leads to very predictable and understandable results. Its drawback is that it is too restrictive. For instance, consider a conflict that arises due to a single file $f$ that is written in $T_h$ and read in $T_i$. Also suppose that $f$ was read within the *SEE* after the time of the last modification operation on $f$ in $T_h$. Then it is clear that $T_i$ used the modified version of $f$ in its computation, and hence it need not be aborted, yet the noninterference criteria will not permit $T_i$ to be committed.

- *Serializability.* This criteria requires that the effect of concurrent transactions be the same as if they were executed in some serial order, i.e., an order in which there was no interleaving of operations from different transactions. In the context of IFS, there are only two possible serial orders, namely, $T_iT_h$ and $T_hT_i$. Serializability has been used very successfully in the context of database transactions, so it is a natural candidate here. However, its use in *SEE* can lead to unexpected results. For instance, consider a situation where a file $f$ is modified in $T_i$ and is deleted in $T_h$. At the point of commit, the user would be looking at the contents of $f$ within the *SEE* and would expect this result to persist after the commit, but if the serial order $T_iT_h$ were to be permitted, then $f$ would no longer be available! Even worse, its contents would not be recoverable. Thus, serializability may be too general in the context of *SEE*: if results were committed automatically when $T_i$ and $T_h$ were serializable, then there is no guarantee that the resulting system state would be as expected by the user of the *SEE*.

- *Atomic execution of SEE activities at commit time.* If the state of main file system after the commit were as if all of the *SEE* activities took place atomically at the point of commit, then it leads to a very understandable behavior. This is because the contents of the main file system after the commit operation will match the contents of the IFS on every file that was read or written within the IFS. The atomic execution criteria (AEC) is a restriction of serializability criterion in that only the order $T_hT_i$ is permitted, and the order $T_iT_h$, which led to unexpected results in the example above, is not permitted.

Based on the above discussion, we use AEC as the criteria for automatic commits in *SEE*. In all other cases, the user will be presented with a set of files and directories that violate the AEC, and the user will be asked to resolve the conflict using one of the options discussed earlier (i.e., abort, redo, or manually reconcile).

In addition to providing consistent results, a commit criteria should be amenable to efficient implementation. In this context, note that we don't have detailed information about the actions within $T_h$. In particular, the UNIX file system maintains only the last read time and write time for each file system object, so there is no way to obtain the list of all read and write actions that took place within $T_h$, or their respective timestamps. We could, of course, maintain such detailed information if we intercepted all file operations on the main file system and recorded them, but this conflicts with our design goal that operations of processes outside *SEE* should not be changed in any way. On the other hand, since we do intercept all file accesses within the IFS, we can (and do) maintain more detailed information about the timestamps of the read and write operations that took place within the *SEE*. Thus, an ideal commit criteria, from an implementation perspective, will be one that leverages the detailed timestamp information we have about $T_i$ while being able to cope with the minimal timestamp information we have about $T_h$. It turns out that AEC satisfies this condition, and hence we have chosen this criteria as the basis for fully automated commits in IFS.

In order to determine whether AEC is satisfied, we need to reason about the timestamps of operations in $T_h$ and $T_i$ and show that their orders can be permuted so that all operations in $T_h$ occur before the operations in $T_i$, and that this permutation does not change the semantics of the operations. We make the following observations in this regard:

- Any changes made within the *SEE* are invisible on the main file system, so the results of operations in $T_h$ would not be changed if all $T_i$ operations were delayed to the point of commit.

- A read operation $R(f)$ performed in $T_i$ can be delayed to the point of commit and still be guaranteed to produce the same results, provided the target $f$ was unchanged between the time $R$ was executed and the time of commit. This translates to requiring that the last modification time of $f$ in the main file system precede the timestamp of the first read operation on $f$ in $T_i$.

- The results of a write operation $W(f)$ performed in $T_i$ is unaffected by any read or write operation in $T_h$, and hence it can be delayed to commit time without changing its semantics.

Based on the observations, we conclude that AEC is satisfied if:

*the earliest read-time of an object within the IFS occurs after the last modification time of the same object on the main file system.*

Note that the latest modification time of an object on the main file system is given by the `mtime` and `ctime` fields associated with that object. In addition, we need to maintain the earliest read-time of every object within the IFS in order to evaluate this criteria.

A slight explanation of the above criteria is useful in the context of append operations on files. Consider a file that is appended by an *SEE* process is subsequently appended by an outside process. Both appends look like a write operation, and hence the above commit criteria would seem to indicate that it is safe to commit results. But if this were done, the results of the append operation performed outside IFS would be lost, which is an unexpected result. Clearly, if the *SEE* process were run at the time of commit, then no information would have been lost. However, this apparent problem is clarified once we realize that an append operation really involves a read and then a write. Once this is taken into account, a conflict will be detected between the time the file was read within IFS and the time it was modified outside, thereby causing the AEC criteria to be violated. More generally, whenever a file is modified within IFS without completely erasing its original contents (which is accomplished by truncating its length to zero), we treat this as a read followed by a write operation for the purposes of committing, and handle the above situation correctly.

**Improvements to AEC** The above discussion of AEC classifies operations into two kinds: read and write. The benefit of such an approach is its simplicity. Its drawback is that it can raise conflicts even when there is a meaningful way to commit. We illustrate this with two examples:

- System log files are appended by many processes. Based on earlier discussion about append operations on files, the AEC criteria won't be satisfied whenever an *SEE* process appends an entry $e_1$ to the log file and an outside process subsequently appends another entry $e_2$ to the same file. Yet, we see that the results can easily be merged by appending both $e_1$ and $e_2$ to the log file.

- Directories close to the root of the file system are almost always examined by *SEE* process as part of looking up a file name in the directory tree. Thus, if any changes were to be made in such directories by outside processes, it will lead to AEC being violated. Yet, we see that a name lookup operation does not conflict with a file creation operation unless the name being looked up is identical to the file created.

These examples suggest that AEC will permit commits more often if we distinguished among operations at a finer level of granularity, as opposed to treating them as read and write operations. However, we are constrained by the fact that we don't have a complete record of the operations executed by outside processes. Therefore, our approach is to try to *infer* the operations by looking at the content of the files. In particular, let $f_o$ denote the (original) content of a file system object at the point it was copied into temporary storage, and $f_h$ and $f_i$ denote the content of the same file in the main file system and the IFS at the point of commit. We can then compute the difference $\delta_h^f$ between $f_o$ and $f_h$, and the difference $\delta_i^f$ between $f_o$ and $f_i$. From these differences, we can try to infer the changes that were made within and outside *SEE*. For instance, if both $\delta_h^f$ and $\delta_i^f$ consist of additions to the end of the file, we can infer that append operations took place, and we can apply these differences to $f_o$.

In the case of directories, the situation is a bit simpler. Due to the nature of directory operations, $\delta_h^f$ will consist of file (or subdirectory) creation and deletion operations. Let $F_h$ denote the set of files created or deleted in $\delta_h^f$, and let $F_i$ be the set of names in this directory that were looked up in $T_i$. This information, as well as the time of first lookup on each of these names, are maintained within the IFS. Let $F_c = F_h \cap F_i$. Now, we can see that the AEC criteria will be satisfied if either one of the following conditions hold:

- $F_c = \phi$, or
- the modification time of $f_o$ precedes all of the lookup times on any of the files in $F_c$.

In the first case, none of the names looked up (i.e., "read") within the *SEE* were modified outside, thus satisfying AEC. In the second case, conflicts are again avoided since all of the lookups on conflicting files took place after any of the modification operations involving them in the main file system.

We point out that inferring operations from the state of the file system can be error-prone. For instance, it is not possible to distinguish from system state whether a file $a$ was deleted or if it was first renamed into $b$ and then deleted. For this reason, we restrict the use of this approach to log files and directories. In other cases, e.g., updates of text files, we can use this technique with explicit user input.

### 3.2.2 Efficient Implementation of Commit

After making a decision on whether it is safe to commit, the next step is to apply the changes to the main file system. One approach in this context is to traverse the contents of the temporary storage and copy them into the main file system. However, this simple approach does not always produce expected results. Consider, for instance, a case where a file $a$ is first renamed to $b$ and then modified. A simple traversal and copy will leave the original version of $a$ as is, and create a new file $b$ whose contents are the same as in the temporary storage. The correct result, which will be obtained if we redo all the (write) operations at the point of commit, will leave the system without the file *a*. Thus, the simple approach for state-based commit does not work correctly.

The above example motivates a log-based solution: maintain a complete log of all successful modifications operations that were performed within the *SEE*, and replay them on the main file system at the point of commit. This approach has the benefit of being simple and being correct in terms of preserving the AEC semantics. However, its drawback is that it is inefficient, both in terms of space and time. In the worst case, the storage overhead can be arbitrarily higher than an approach that uses state-based committing. For instance, consider an application that creates and deletes many (temporary) files. The state-based approach will need to store very few files in temporary storage, but a log-based approach will need to store all the write operations that were performed, including those on files that were subsequently deleted. Moreover, redoing the log can be substantially more expensive than state-based commit, since the latter can exploit rename operations to avoid file copies altogether.

The above discussion brings forth the complementary benefits of the two approaches. The first approach makes use of the accumulated modification results on file system objects, thus avoiding the expense associated with the maintenance and redoing of logs. The second approach, by maintaining logs, is able to handle subtle cases involving file renames. In our implementation of the commit operation, we combine the benefits of both.

We refer to our approach as state-based commit. For files, the commit action used in our approach involves simply renaming (or copying) the file into the main file system. For operations related to links, it records a minimal set of link-related operations that captures the set of links associated with each file system object. In this sense, one can think of the approach as maintaining "condensed" logs, where redundant information is pruned away. For instance, there is no need to remember operations on a file if it is subsequently deleted. Similarly, if a file is renamed twice, then it would be enough to remember the net effect of these two renames. To identify such redundancies efficiently, our approach partitions the logs based on the objects to which they apply. This log information is kept in the inode table described earlier.

Operations that modify the contents of a file or change metadata (such as permissions) on any file system object are not maintained in the logs, but simply applied to the object. In effect, the state of the object captures the net effect of all such operations, so there is no need to maintain them in a log. Thus, only information about file or directory creation and deletion, and those that concern addition or removal of links are maintained in the log. In addition, to simplify the implementation, we separate the effects of creating or deleting file system objects from the effect of adding or deleting links. This means that the creation of a file would be represented in our logs by two operations: one to create the file object, and another to link it to the directory in which the object is created. Similarly, a rename operation is split into an operation to add a link, another to remove a link, and a third (if applicable) to delete the file originally referenced by the new name. As in previous sections, file objects involved in these operations are identified by inode numbers rather than path names.

Specifically, the log contains one of the following operations:

- *create* and *delete* operations denote respectively the creation of a file or a directory, and are associated with the created file system object.
- *addlink* and *rmlink* operations denote respectively the addition and deletion of a link from a directory to a file system object. These operations are associated with the file system object that is the target of the link, and have two operands. The first is the inode number of the parent directory and the second is the name associated with the link.

The effect of some of these operations is superceded by other operations, in which case only latter operations are maintained. For instance, a delete operation supercedes a create operation. An rmlink operation cancels out a preceding addlink with the same operands.

In addition to removing redundant operations from the logs, we also reorder operations that do not interfere with each other in order to further simplify the log. In this context, note that two valid addlink operations in the log associated with any file system object are independent. Similarly, any addlink operation on the object is independent of an rmlink operation. (Both these statements are true only when we assume that operations that are superceded or canceled by others have already been removed from the log.)

Based on this discussion, we can see that a condensed log associated with a file system object can consist of operations in the following order:

- zero or one create operation. Since the file system object does not exist before creation, this must be the first operation in the log, if it exists.

- zero or more rmlink operations. Note that multiple rmlink operations are possible if the file system object was originally referenced by multiple links. Moreover, the parent directories corresponding to these rmlink operations must all have existed at the time of creation of *SEE*, or otherwise an addlink operation (to link this object to the parent directory) must have been executed before the rmlink. In that case, the addlink and rmlink operations would have cancelled each other out and hence won't be present in the condensed log.

- zero or more addlink operations. Note that multiple addlink operations are possible if the object is being referenced by multiple links. Also, there must be at least one addlink operation if the first operation in the log is a create operation.

- zero or one delete operation. Note that when a delete operation is present, there won't be any addlink operations, but there may be one or more rmlink operations in the log.

Given the condensed logs maintained with the objects in the inode table, it seems straightforward to carry out the commit operation. The only catch is that we only have the relative ordering of operations involving a single file system object, but lost information about the global ordering of operations across different objects. This raises the question as to whether the meanings of these operations may change as a result. In this context, we make the following observations:

- Creation and deletion operations do not have any dependencies across objects. Hence the loss of global ordering regarding these operations does not affect the semantics of these operations.

- Rmlink operation depends upon the existence of parent directory, but nothing else. This means that as long as it is performed prior to the deletion of parent directory, its meaning will be the same as it was executed in the global order in which it was executed originally.

- Addlink operation depends on the creation of the parent directory (i.e., the directory in which the link will reside) and the target object. Moreover, an addlink operation involving a given parent directory and link name has a dependency on any other rmlink operation involving the same parent directory and link names. This is because the addlink operation cannot be performed if a link with the same name is present in the parent directory, and the execution of rmlink affects whether such a link is present. Thus, the effect of addlink operations will be preserved as long as any parent directory creation, as well as relevant rmlink operations are performed before.

Among operations that have dependency, one of the two possible orders is allowable. For instance, an rmlink operation cannot precede the existence of either the parent directory or the target of the link. Similarly, an addlink operation cannot precede an rmlink operation with the same parent directory and name components. (Recall that we have decomposed a rename operation into rmlink (if needed), addlink and an object delete (if needed) operations, so it cannot happen that an addlink operation is invoked on a parent directory when there is already another link with the same name in that directory.) This means that even though the global ordering on operations has been lost, it can be reconstructed. Our approach is to traverse the file system within the temporary storage, and combine the condensed logs while respecting the above constraints, and then execute them in order to implement the commit step.

*Atomic Commits.* As mentioned before, the committing of modifications should be done atomically in order to guarantee file system consistency. The natural way to do atomic operations is through file-locking: to prevent access to all the file system objects that are to be modified by the committing process. We use Linux mandatory locks to achieve this. Immediately before the committing phase, a lock is applied to the list of to-be-committed files, so that other processes do not gain access to these files. Only when the committing is completely done, the locks on these files are released.

## 3.3 Discussion

In the previous two sections, we discussed aspects of IFS, our filesystem proxy. In this section, we discuss how the other components of *SEE* fit together, including the components that support restriction, network level redirection, and user interface.

### 3.3.1 Implementing Restriction at System Call Layer.

The actions of *SEE* processes are regulated by a kernel-resident policy enforcement engine that operates using *system call interposition.* This enforcement engine generally enforces the following policies in order to realize *SEEs*:

- *File accesses.* Ensure that *SEE* processes can access only the files within the IFS. Access to device special files are not allowed, except for "harmless" devices like `tty`'s and `/dev/null`.

- *Network access.* Network accesses for which an explicit (static) redirection has been set up are allowed. The redirection may be to another process that executes within a different *SEE*, or to an intelligent proxy for a network service. (Note that network file access operations do not fall in this category — they are treated as file operations.)

- *Interprocess communication (IPC).* IPC operations are allowed among the processes within the same *SEE*. However, no IPC may take place between *SEE* and non-*SEE* processes. An exception to this rule is currently made for X-server access. (To be safe, we should restrict X-server accesses made by *SEE* applications so that they don't interfere with X-operations made by non-*SEE* applications. However, our implementation does not currently have the ability to enforce policies at the level of X-requests.)

- *Signals and process control.* A number of operations related to process control, such as sending of signals, are restricted so that a process inside an *SEE* cannot interfere with the operation of outside processes.

- *Miscellaneous "safe" operations.* Most system calls that query system state (timers and clocks, file system statistics, memory usage, etc.) are permitted within the *SEE*. In addition, operations that modify process-specific resources such as timers are also permitted.

- *Privileged operations.* A number of privileged operations, such as mounting file systems, changing process scheduling algorithms, setting system time, and loading/unloading modules are not permitted within *SEE*.

Note that the exact set of rules mentioned above may not suit all applications. For instance, one may want to disallow all network accesses for an untrusted application, but may be willing to allow some accesses (e.g, DNS and WWW) for applications that are more trusted. To support such customization, we use a high-level, expressive policy specification language called BMSL [58, 66] in our implementation. This language enables convenient specification of policies that can be based on system call names as well as arguments. The kinds of policies that can be expressed include simple access control policies, as well as policies that depend on history of past accesses and/or resource usage. In addition, the language allows response actions to be launched when policies are violated. For instance, it can be specified that if a process tries to open a file $f$, then it should be redirected to open another file $f'$. Efficient enforcement engines are generated by a compiler from these policy specifications. More details about this language and its compiler can be found in [66].

In our experience, we have been able to specify and enforce policies that allow a range of applications to function without raising exceptions, and the *SEE* experimentation chapter describes some of our experiences in this regard.

### 3.3.2 Support for Network Operations.

Support for network access can be provided while ensuring one-way isolation semantics in the following cases:

- access to services that only provide query (and no update) functionality, e.g., access to domain name service and informational web sites, can be permitted by configuring the kernel enforcement engine so that it permits access to certain network ports on certain hosts.

- communication with processes running within other *SEEs* can be supported by redirecting network accesses appropriately. This function is also provided by the kernel enforcement engine.

- accesses to any service can be allowed, if the access is made through an intelligent proxy that can provide isolation semantics.

Currently, our implementation supports the first two cases. Use of distributed *SEEs* provides an easy way to permit isolated process to access any local server — one can simply run the server in isolation, and redirect accesses by the isolated process to this isolated server. However, for servers that operate in a different administrative domain, or servers that in turn access several other network functions, running the server in isolation may not always be possible. In such cases, use of an intelligent proxy that partially emulates the server function may be appropriate.

Intelligent proxies may function in two ways. First, they may utilize service-specific knowledge in filtering requests to ensure that only "read" operations are passed on to a server. Second, they may provide some level of support for "write" operations, while containing the effects within themselves, and propagating the results to the real server only at the point of commit. For instance, an email proxy may be implemented which simply accepts email for delivery, but does not actually deliver them until commit time. Naturally, such an approach won't work in the case when a response to an email is expected.

Another limitation of our current implementation is that it does not provide support for atomic commits across distributed *SEEs*.

### 3.3.3 User Interface.

In this section, we describe the support provided in our implementation for users to make decisions regarding commits.

Typically, an *SEE* is created with an interactive shell running inside it. This shell is used by the user to carry out the tasks that he/she wishes to do inside the *SEE*. At this point, the user can use arbitrary helper applications to analyze, compare, or check the validity of the results of these tasks. For instance, if the application modifies just text files, utilities like `diff` can point out the differences between the old and new versions. If documents, images, video or audio files are modified, then corresponding document or multimedia viewers may be used. More generally, users can employ the full range of file and multimedia utilities or customized applications that they use everyday to examine the results of *SEE* execution and decide whether to commit.

Before the user makes a final decision on committing, a compact summary of files modified within the *SEE* is provided to the user. If the user does not accept the changes, she can just roll them back at a click of button. If she accepts the changes, then the commit criteria is checked. If it is satisfied, then the commit operation proceeds as described earlier. If not, the user may still decide to proceed to commit, but this is supported only in certain cases. For instance, if the whole structure of the file system has been changed outside the *SEE* during its operation, there won't be a meaningful way to commit. For this reason, overriding of commit criteria is permitted only when the conflict involves a plain file.

Recall that *SEEs* may be used to run untrusted and/or malicious software. In such cases, additional precautions need to be taken to ensure that this software does not interfere with the helper applications, subverting them into providing a view of system state that looks acceptable to the user. In particular, we need to ensure that untrusted processes cannot interfere with the operation of helper application processes, or modify the executables, libraries or configuration files used by them. To ensure this, helper applications can be run outside of the *SEE*, but having a read-only access to the file system view within the IFS using a special path name. This approach ensures that the helper application gets its executable, libraries and config files from the host file system. Another advantage of doing this is that any modifications to the system state made by helper applications do not clutter the user interface that reports file modifications that were carried out within the *SEE*. (While it may seem that helper applications are unlikely to modify files, this is not true. For instance, running the bash shell causes it to update the `.bash_history` file; running a browser updates its history and cache files; and so on.)

# CHAPTER 4

# Evaluation of *SEE*

In this chapter, we present an evaluation of the functionality and performance of our *SEE* implementation.

## 4.1 Evaluation of Functionality

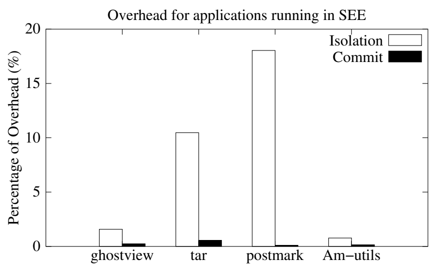
**Untrusted applications.** We describe two applications here: a file renaming utility freeware called `rta` [65], which traverses a directory tree and renames a large number of files based on rules specified on the command line, and a photo album organizer freeware called `picturepages` [45]. These applications ran successfully within our *SEE*. Our implementation includes a GUI that summarizes files modified in the *SEE* so as to simplify user's task of deciding whether the changes made by the application are acceptable. Using this GUI, we checked that the modifications made by these applications were as intended: renaming of many files, and creation of several files and/or directories. We were then able to commit the results successfully.

To simulate the possibility that these programs could be malicious, we inserted an attack into `picturepages` that causes it to append a new public key to the file `.ssh/authorized_keys`. (This attack would enable the author of the code to later log into the system on which `picturepages` was run.) Using our GUI, it was easy to spot the change to this file. The run was aborted, leaving the file system in its original state.

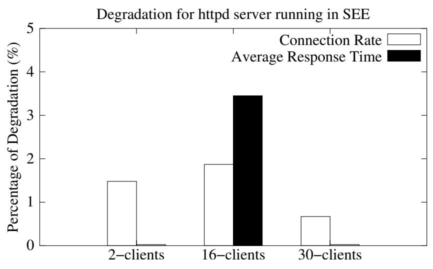
**Malicious code.** Email attachments and WWW links are a common source of viruses and other malware. We used an *SEE* to protect systems from such malware. Specifically, we modified the MIME type handler configuration file used by Mozilla so that executables, as well as viewers launched to process documents (e.g., `ghostscript` and `xpdf`) fetched over the Internet, were run within *SEE*. We fetched sample malicious PostScript and Perl code over the network using this approach. This code was executed inside the *SEE*. Using our GUI, we were able to see that these programs were performing unexpected actions, e.g., creating a huge file in the user's home directory. These actions were aborted. Also, at the time of implementing this approach, there were several image flaw exploits (JPEG virus) that have captured the attention of many researchers. Running such image viewers inside an *SEE* will help eliminate this potential danger, because any malicious activity from the exploits will be isolated from affecting the main system.

Some kinds of malicious code are written to recognize typical sandbox environments, and if so, not display their malicious behavior. This can cause a user to develop trust in the code and then execute it outside of sandbox, when the malcode will deliver its payload. With our approach, we point out that running the code inside *SEE* does not incur significant inconvenience for the user, thereby making it easy for the user to always use it. In this case, the code will always display benign behavior.

**Software installation.** Another experiment performed a trial installation of `mozilla` browser. During the installation, an incorrect directory name `/usr/bin` was chosen as the location for installation, instead of the default directory `/usr/local/mozilla`. Under normal circumstances, this causes Mozilla to copy a number of files into `/usr/bin`, thereby "polluting" the directory. After running the program in an *SEE*, the user interface indicated that a large number of files (some are non-executables) were added to `/usr/bin`, which was not desirable. Aborting this installation, we ran the installation program a second time, this time with `/usr/local/mozilla` as the location for installation. At the end of installation, we restarted the browser, and visited several sites



(a) Utility applications



(b) Apache httpd server

Figure 3: Performance Results for Program Execution in *SEE*

to make sure that the program worked as expected. (For this experiment, the system call restriction layer was modified to allow all WWW accesses.) Finally, we committed the installation, and from that point on, we were able to use the new installation of the browser successfully, outside of *SEE*.

**Upgrading and testing a server.** Specifically, we wanted to upgrade our web server so that it can support SSL. We started a command shell under *SEE*, and used it to upgrade the apache software installation. We then ran the new server. To enable it to run, we used static redirection for network operations, so that a bind operation to port 80 was redirected to port 3080. We then ran a browser that accessed this server by connecting to this port. We verified that the new server worked correctly. Meanwhile, the original server was still accessible to every one. Thus, *SEE* allowed the software upgrade to be tested easily and conveniently, without having to shutdown the original server.

After verifying the operation of the new server, we attempted to commit the results. Unfortunately, this produced conflicts on some files such as the access and error log files used by the server. We chose to ignore updates to such output files that were made within the *SEE*, and commit only the rest of the files, which could be done successfully.

## 4.2 Evaluation of Performance

All performance results reported in this part were obtained from a laptop running Red Hat Linux 7.3 with a 1.0GHz AMD Athlon4 processor, 512MB memory and a 20GB, 4200rpm IDE hard disk. The primary metric was elapsed time.

For performance evaluation, we considered the following classes of examples:

- *Utility programs.* In this category, we studied `ghostview` and `tar` utilities. Specifically, we performed ghostview on a 31M file, with no file modification operations; and `tar` to generate a tarball from a 26M directory, and the only modification operations involved was the creation of this archive. From Figure 3, we can see a 3-12% overhead incurred for such applications during isolation phase, and a negligible commit time overhead.

- *Servers.* We measured the performance overhead on the Apache web server using WebStone [70], a standard web server benchmark. We used version 2.5 of this benchmark, and ran it on a separate computer that is connected to the server through a 100Mbps network. We ran the benchmark with two, sixteen and thirty clients. In the experiments, the clients were simulated to access the web server concurrently. They randomly fetch html files whose size is from 500 bytes to 5M. The benchmark was run for a duration of 30 minutes, and the results were averaged across ten such runs. The results are shown in Figure 3.

- *File system benchmarks.* We used `Postmark` [27] and `Am-Utils` [43] benchmarks to get the benchmark data for IFS. Postmark is a file system benchmark to measure the performance for file system used by Internet applications, such as email. In this experiment, we configured `Postmark` to create 500 files in a file pool, with file sizes ranging from 500 bytes to 500KB. A total of over 5000 file system operations were performed. In total, 1515 files were created, 1010 files read, 990 file written, and 1515

|  | Log-based Commit | State-based Commit |  |
|---|---|---|---|
|  | Time | Time | Speedup |
| ghostview | 0.03 | 0.03 | 1 |
| tar | 0.14 | 0.03 | 4.7 |
| postmark | 225 | 0.07 | 3214.3 |
| Am-utils | 16.9 | 0.35 | 48.3 |

Table 1: Comparison for Log-based Commit and State-based Commit. All numbers are in seconds.

files deleted. The tests were repeated ten times. The results are as depicted in figure 3. Overall, a 18% performance degradation is observed, and commit overhead is near zero. `Am-Utils` is a CPU-intensive benchmark result by building the Am-Utils package, which contains 7.6M lines of C code and scripts. The building process creates 152 files and 19 directories, as well as 6 rename and 8 setattr operations. We ran this experiment in both original file system and IFS. The results, shown in Figure 3, indicate a low isolation overhead of under 2% and a negligible commit overhead.

In addition, we also collected results in Table 1 to show the efficiency of our state-based commit approach. An implementation that used log based committing was compared with our state based committing implementation, and the performance of both of the approaches were compared for applications such as `tar`, `postmark` and `Am-utils`. The results project the advantage of using a state based commit approach, particularly illustrating the advantage of having accumulative effects for file objects. For instance, the large number of temporary files created then deleted in Am-utils compilation and all the files created then deleted in Postmark execution, are not considered in the committing stage as candidates, while log-based commit still needs to perform the whole set of operations (e.g. write) to all these files, so there is a significant difference between the two approaches in terms of commit time.

# CHAPTER 5

# Related Work of *SEE*

**Sandboxing.** Sandboxing based approaches [20, 15, 4, 47, 56, 48] involve observing a program's behavior and blocking actions that may compromise the system's security. Janus [20] incorporates a `proc` file system based system call interposition technique for the Solaris operating system. A more recent version has been implemented on Linux, and uses a kernel module for interposition. Chakravyuha [15] uses a kernel interception mechanism. MAPbox [4] is a sandboxing mechanism where the goal is to make the sandbox more configurable and usable by providing a template for sandbox policies based on a classification of application behaviors. [47] creates the policy sandbox for programs (such as web browser) by first tracking the file requests made by the programs. This approach, however, requires a training phase, in which users need to run the programs using "normal" inputs, so that the policy sandbox can capture a complete set of files accessed by the programs. But in the case of untrusted code, the choice of such inputs may not be clear. Safe Virtual Execution (SVE) [56] uses Software Dynamic Translation, a technique for modifying binaries as they execute, to implement sandboxing. Systrace [48] is a sandboxing system that notifies the user about all system calls that an application tries to execute and then uses the response from the user to generate a policy for the application.

The main drawback of sandboxing based approaches is the difficulty of *policy selection*, i.e., determining what actions are permissible for a given piece of software. Note that malicious behavior may not only involve accessing unauthorized resources, but also accessing authorized resources in unauthorized ways. For instance, a program that creates a compressed version of a file may instead create a file that contains no useful data, which is equivalent to deleting the original file. It is unlikely that a practical system can be developed that can allow users to conveniently state policies that allow write access to the file while ensuring that the file is replaced with its compressed version. In contrast, an *SEE* permits manual inspection, aided by helper applications, to be used to determine if a program behaved as expected. This approach is much more flexible. Indeed, it is hard to imagine that tasks such as verifying whether a software package has been installed properly can even be formally specified using any sandbox-type policy.

[57, 77] extend sandboxing by allowing operations to be disallowed silently, i.e., by returning a success code to the program. The goal of the approaches is deception, i.e., making a malicious program believe that it is succeeding in its actions so as to observe its behavior. In our terminology, these approaches use restriction rather than redirection. As we observed earlier, use of restriction is likely to break many benign applications, as well as alert malicious applications very quickly to the effect that their actions are not succeeding. For instance, if a write operation is silently suppressed, the application can easily detect this by reading back the contents.

**Isolation approaches.** Two-way isolation between a host and guest operating system forms the basis of security in virtual machine based approaches for realizing *SEEs*. The "playground" approaches developed for Java programs in [36, 12] also belong to this general category — untrusted programs are run on a physically isolated system, while their display is redirected to the user's desktop. Note that the file system on the user's computer cannot directly be accessed on the playground system, which means that there is two way isolation being employed in this case. Covirt [11] proposes that most of applications be run inside virtual machine instead of host machines. Denali [71] is another virtual machine based approach that runs untrusted distributed server applications. As outlined in the introduction, all the above approaches suffer from the difficulty of environment reproduction, and also in committing the changes back to the original system.

[33] was the first approach to present a systematic development of the concept of *one-way isolation* as an effective means to isolate the effects of running processes from the point they are compromised. They developed protocols for realizing one-way isolation in the context of databases and file systems. However, they do not present an implementation of their approach. As a result, they do not consider the research challenges that arise due to the nature of COTS applications and commodity OSes. Moreover, they do not provide a systematic treatment of issues related to consistency of committed results.

[32] developed a practical approach for secure execution of untrusted software based on isolation. The focus of this effort was on developing a tool that can be easily installed and used by ordinary users that may not have administrative access to a computer. It is implemented entirely at the user level, and does not require any changes to the OS kernel. In order to achieve this objective, [32] compromises on performance as well as generality. In particular, the approach suffers from high overheads that can be over 100% in some instances. Moreover, isolation semantics cannot be faithfully reproduced for certain operations that involve meta-data such as permissions and ownership. For directories, isolation is achieved using an ad-hoc approach that is hard to implement and its semantics is inconsistent with that of file updates. Finally, no systematic solution to the commit problem is provided. The approach developed in this part addresses all these drawbacks by implementing isolation within the kernel at the VFS layer. Moreover, it shows how the approach can be generalized so that isolation can be provided for non-file operations, e.g., certain classes of network accesses.

On the other hand, Nooks [63] focuses on improving OS reliability by isolating OS from buggy device drivers. This can be deemed complementary to our work, as it targets kernel land untrusted drivers, while our approach targets at isolating user land untrusted programs.

**Recovery-oriented systems.** The Recovery-Oriented Computing (ROC) project [51] develop techniques for fast recovery from failures, focusing on failures due to operator errors. [10] presents an approach that assists recovery from operator errors in administering a network server, with the specific example of an email server. The recovery capabilities provided by their approach are more general than those provided by ours. The price to be paid for achieving more general recovery capabilities is that their implementation needs to be application specific, and hence will have to be tailored for each specific application/service. In contrast, we provide an application-independent approach. Another important distinction is that with our approach, consistency of system state can be assured whenever the commit proceeds successfully. With the ROC approach, which does not restrict network operations, there is no way to prevent the effects of network operations from becoming so widely distributed in the network they cannot be fully reversed. In the case of email service, they allow a certain level of inconsistency, e.g., redelivering an email that was previously read and deleted by a client, and expect the user to manually resolve this inconsistency. This potential for inconsistency is traded in favor of eliminating the risk of commit failures.

**File system approaches.** Elephant file system [54] is equipped with file object versioning support, and supports flexible versioning policies. [13, 50, 52, 62, 44] use check pointing technique to provide data versioning. [39] implements VersionFS, a versatile versioning file system. They use a stackable template file system as ours, and use a sparse file technique to reduce storage requirements for storing versions of large files. While all of these approaches provide the basic capability to rollback system state to a previous time, such a rollback will discard *all* changes made since that time, regardless of whether they were done by a malicious or benign process. In contrast, the one-way isolation approach implemented in this part guarantees selective rollback of the actions of processes run within the *SEE* without losing the changes made by benign processes executing outside of the *SEE*.

Repairable File System [80, 79] makes use of versioning file system to bring repair facility to a compromised file server. Fastrek [46] applies the similar approach to protect databases. These approaches can attribute changes to malicious or benign process executions, and allow a user to rollback changes selectively. However, since the changes made by (potentially) compromised processes are not contained within any environment, "cascading aborts" can become a problem. Specifically, a benign process may access the data produced by a compromised process, in which case the actions of the benign process may have to be rolled back, as well as the actions of processes that used the results of such a benign process and so on. The risk of such cascaded aborts should be weighed against the risk of not being able to commit in our approach. Thus, this approach as well as the ROC approach mentioned above are more suitable when the likelihood of rollbacks is low, and commit failures cannot be tolerated.

Loopback file system [34] can create a virtual file system from existing file system and allow access to existing files using alternative path name. But this approach provides no support for versioning or isolation.

3D file system [28] provides a convenient way for software developers to work with different versions of a software package. In this sense, it is like a versioning file system. It also introduces a technique called *transparent viewpathing* which is based on translating file names used by a process. It gives a union view of several directory structures thus allowing an application to transparently access one directory through another's path. As it is not designed to deal with untrusted applications, it needs the cooperation from the application for this mechanism to work. TFS [64] is a file system in earlier distributions of Sun's operating system (SunOS), which allowed mounting of a writable file system on top of a read-only file system. TFS also has a view similar to 3DFS, where the modifiable layer sits on top of the read only layers. [42] describes a union file system for BSD, that allows "merging" of several directories into one, with the mounted file system hiding the contents of the original directories. The union mount will show the merger of the directories and only the upper layer can be modified. All these file systems are intended for software development, with the UnionFS providing additional facilities for patching read only systems. However, they do not address the problem of securing the original file system from untrusted/faulty programs; nor do they consider problems such as data consistency and commit criteria.

# Part II

# Practical Proactive Integrity Preservation

# CHAPTER 6

# Overview of *PPI* Approach

Today's malware defenses rely mainly on reactive approaches such as signature-based scanning, behavior monitoring, and file integrity monitoring. Unfortunately, attackers can easily modify the structure and behavior of their malware to evade detection by signature-based or behavior-based techniques. They may also subvert system integrity monitoring tools using rootkit-like techniques. It is therefore necessary to develop proactive techniques that can stop malware before it damages system integrity.

Sandboxing is a commonly deployed proactive defense against untrusted (and hence potentially malicious) software. It restricts the set of resources (such as files) that can be written by an untrusted process, and also limits its communication with other processes on the system. However, techniques that regulate write-access without restricting read-access aren't sufficient to address adaptive malware threats. Specifically, they do not satisfactorily address indirect attacks, where a benign application ends up consuming malware outputs stored in persistent storage (e.g., files). For instance, malware may modify the following types of files used by a benign application:

- *System libraries, configuration files or scripts.* One may attempt to eliminate this possibility by preventing untrusted software from storing any files in system directories, but this will preclude the use of many legitimate (untrusted) applications that expect to find their binaries, libraries and configuration files in system directories. Alternatively, one can explicitly enumerate all the files in system directories that are used by benign applications, but this becomes a challenging task when we consider the number of such files — for instance, a typical desktop Linux distribution contains over 100K files in system directories. Errors may creep into such enumerations, e.g., one may leave out optional libraries (e.g., application extensions such as Apache modules, media codecs, etc.) or configuration/customization files, thereby introducing opportunities for indirect attacks.

- *User-specific customization files and scripts.* Identifying all user-specific scripts and customization files is even harder: different applications use different conventions regarding the location of user-specific customization files. Moreover, some of these files may in turn load other user files, or run scripts within user directories. Static identification of all the files used by a benign application may be very hard.

  We observe that significant harm can result from unauthorized modifications to user files. For instance, by altering ssh keys file, malware may enable its author to log into the system on which it is installed. By modifying a file such as `.bashrc`, e.g., by creating an alias for a command such as `sudo`, malware can cause a Trojan program to be run each time `sudo` is used. Worse, malware can first modify a configuration file used by a less conspicuous benign application, such as a word-processor. For instance, it may replace the name of a word-processor plug-in with a Trojan program that in turn modifies `.bashrc`.

- *Data files.* Malware may create data files with malicious content that can exploit vulnerabilities in benign software. The recent spate of vulnerabilities in image and document viewers, web browsers, and word-processors shows that this is indeed a viable approach. Malware may save these files where they are conspicuous (e.g., on the desktop), using names that are likely to grab user attention. When the user invokes a benign viewer on the file, it will be compromised. At this point, malware can achieve its objectives using the privileges available to the viewer.

In contrast, we develop an approach in this part that aims to *provide positive assurance about overall system integrity.* Our method, called *PPI* (Practical Proactive Integrity protection), identifies a subset of objects (which are typically files) as *integrity-critical* and a set of *untrusted* objects. We assume that system integrity is preserved as long as untrusted objects are prevented from influencing the contents of integrity-critical objects either directly (e.g., by copying of an untrusted object over an integrity-critical object) or indirectly through intermediate files. In other words, there should be no information flow from untrusted objects to integrity-critical objects.

Although information-flow based integrity preservation techniques date as far back as the Biba integrity model [7], these techniques have not had much success in practice due to two main reasons. First, these techniques require every object in the system to be labeled as high-integrity or low-integrity — a cumbersome task, considering the number of files involved (more than 100K on typical Linux systems). Manual labeling is prone to errors that can either damage system integrity (by allowing an integrity-critical file to be influenced by a low-integrity application) or usability (by denying a legitimate operation as a result of security violation). Secondly, the approach is not very flexible, and hence breaks many applications. To overcome this problem, many applications may need to be designated as "trusted," which basically exempts them from the information flow policy. Obviously, an increase in the number of trusted applications translates to a corresponding decrease in assurance about overall integrity.

As a result of the factors mentioned above, information-flow based techniques have not become practical in the context of contemporary operating systems such as Windows and Linux. In contrast, we have been able to develop a practical information-flow based integrity protection for desktop Linux systems by focusing on (a) automating the development of integrity labels and policies, (b) providing a degree of assurance that these labels and policies actually protect system integrity, and (c) developing a flexible framework that can support contemporary applications while minimizing usability problems as well as the need to designate applications as "trusted." Our experiments considered a modern Linux Workstation OS together with numerous benign and untrusted applications, and showed that system usability is preserved by our technique, while thwarting sophisticated malware.

## 6.1 Goals of Approach

The objectives of our approach include the following:

- *Provide positive assurances about system integrity* on a contemporary Workstation, e.g., a Linux CentOS/Ubuntu desktop consisting of hundreds of benign applications and tens of untrusted applications. *Integrity should be preserved even if untrusted programs run with root privileges.*

- *Effectively block rootkits and most other malware.* Most malware, including rootkits and spyware, should be detected when they attempt to install themselves, and removed automatically and cleanly. Stealthier malware should be detected when they attempt to damage system integrity, and can be removed at that point. *We do not address malware that can operate without impacting system integrity,* e.g., a P2P application that transmits user data to a remote site when it is explicitly invoked by a user.

- *Be easy to use,* for end-users as well as system administrators. Usability encompasses the following:

  - *Preserve availability of benign applications,* specifically, provide a certain level of confidence that benign applications would not fail due to security violations during their normal use.
  - *Minimize administrator effort* by automating the development of file labels and integrity policies.
  - *Eliminate user prompts.* Security mechanisms that require frequent security decisions from users don't work well in practice for two

reasons. First, users get tired and don't cooperate. Second, these user interactions become the basis for social engineering attacks by malware writers.

- *Reduce reliance on trusted applications* so as to provide better overall assurance.

## 6.2 Salient Features

The principal elements of our approach that enables us to achieve the above goals are summarized below:

- *Flexible decomposition of high-level policies into low-level policies.* Traditional approaches for information-flow based integrity, such as the Biba integrity model [7], associate a label with an object (or subject) at the time of their creation, and this label does not change during the lifetime of the object or subject. In other words, these labels effectively define the access policies: a subject is permitted to read (write) an object only if the subject's integrity is equal to or lower (equal to or higher) than that of the object. In contrast, we distinguish between *labels,* which are a judgment of the trustworthiness of an object (or subject), from *policies* that state whether a certain read or write access should be permitted. Based on this separation, our approach (described in Section 7.1) allows integrity levels of objects or subjects to change over their lifetime. Moreover, "read-down" and "write-up" conflicts are resolved differently for different objects and subjects. These factors provide flexibility in developing low-level policies that preserve system integrity without unduly impacting usability.

- *Automated analysis for generating enforceable policies.* Given the large number of objects (hundreds of thousands) and subjects (thousands), manual setting of policies for every object/subject pair is impractical. In Section 7.2, we therefore develop techniques that utilize an analysis of access patterns observed on an unprotected system to automatically derive policies. This analysis can also be used to automatically complete the set of integrity-critical applications, starting from a partial list provided by a policy developer.

  As we show in Section 7.2.3, our technique is sound, i.e., it will generate policies that preserve system integrity, even if the access logs used in analysis are compromised by malicious applications running on the unprotected system. However, corrupted logs can compromise system availability.

- *A flexible enforcement framework.* Our enforcement framework, described in Section 7.4, consists of a small, security-critical enforcement component that resides in the OS kernel, and a user-level component that incorporates more complex features that enhance functionality without impacting security. This framework also incorporates features needed for learning and synthesizing policies for new applications.

- *Mechanisms for limiting trust.* There are some instances when high-integrity applications should be allowed to access low-integrity files. In Section 7.3, We develop techniques that enable such exceptions to be restricted. Our techniques typically have the effect of distinguishing between code/configuration inputs from data inputs, and ensuring that exceptions are made only for data inputs. Using these mechanisms, we describe how we can limit the amount of trust placed on important applications such as software installers, web browsers and email handlers, and file utilities.

We have implemented our technique on desktop systems running RedHat/Ubuntu Linux, consisting of several hundred benign software packages and a few tens untrusted packages, the evaluation shows that the approach is practical, and does not impose significant usability problems. It is also effective in preventing installation of most malware packages and detection (and blocking) of malicious actions performed by stealthy malware.

# CHAPTER 7

# Design and Implementation of *PPI*

In this Chapter, we present how we design and implement different components of *PPI* as outlined in Chapter 6.

## 7.1 Policy Framework

### 7.1.1 Trust and Integrity Levels

Figure 4 illustrates the integrity and trust levels used in our framework. To simplify the presentation, we use just two integrity levels: *high* and *low*. Integrity labels are associated with all objects in the system, including files, devices, communication channels, etc. A subset of high-integrity objects need to be identified as integrity-critical, which provide the basis for defining system integrity:

**Definition 1 (System Integrity)** *We say that system integrity is preserved as long as all integrity-critical objects have high integrity labels.*

The set of initial integrity-critical objects is externally specified by a system administrator, or better, by an OS distribution developer. It is important to point out that the integrity-critical list need not be comprehensive: if objects

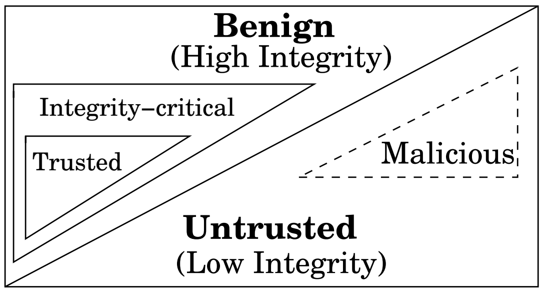


Figure 4: Classification of Applications in *PPI* Policy Framework

are left out of this list, our technique will automatically detect them using the analysis described in Section 7.2, as long as these objects are accessed after the kernel module enforcing *PPI* policies has begun execution. In particular, objects that are accessed during early phases of the boot process, as well those objects that are accessed by *PPI*, must be explicitly included in the integrity-critical set. On Linux, this (minimal) set of integrity-critical objects includes all the files within `/boot`, the binaries used by `init` and the files used by *PPI*, as well as devices such as the hard disk.

When referring to applications, we use the term “trust level” instead of “integrity level.” Benign applications are those that are known to be benign, and may include applications that are obtained from a trusted source such as an OS vendor. Hence the files that constitute benign applications will have high integrity. Moreover, benign applications will remain trustworthy (i.e., produce high-integrity outputs) as long as they are never exposed to low-integrity (and hence potentially malicious) inputs. A subset of benign applications may be designated as *trusted.* These applications are deemed to sufficiently validate their inputs that they can produce high-integrity outputs even when some of their inputs have low-integrity. Untrusted applications are those that are obtained from untrusted sources, e.g., downloaded from an unknown website or those arriving via email from unauthenticated sources. An unspecified subset of these applications may be malicious.

Since trusted applications are being exempted from information flow policies, it is important that only a small number of well-tested applications are designated this way. In addition, the scope and purpose of this trust should be minimized as much as possible. We defer these issues until Section 7.3.

### 7.1.2 Integrity Labels Versus Policies

Given our goal of preserving system integrity, the Biba model is an obvious starting point [7]. However, a traditional interpretation of multi-level security (MLS) can lead to a rigid system that is difficult to use. To address this problem, we distinguish between *integrity labels* and *policies*. In our view, an integrity label on a file simply indicates whether its content is trustworthy, but does not dictate whether trustworthiness must be preserved, say, by preventing low-integrity subjects from writing to that file. A policy, on the other hand, indicates whether a certain read or write access should be permitted. This separation yields flexibility in developing policies that preserve system integrity without unduly impacting usability. For instance, we have the following choices for policies when a high-integrity subject (process) attempts to read a low-integrity object:

- **deny:** deny the access
- **downgrade:** downgrade the process to low-integrity
- **trust:** allow the access without downgrading the process, trusting the process to protect itself

The following examples illustrate the need for this flexibility. Consider a utility such as `cp` that accesses high-integrity objects in some runs (e.g., copy `/etc/passwd`), and accesses low-integrity objects in other runs (e.g., copy user files). Downgrading (the second alternative above), which corresponds to the low-water mark (LOMAC) policy [7, 8, 19], permits such dual use. However, this choice of downgrading is inappropriate in some cases, and leads to the well-known *self-revocation* problem: consider a process that has opened a high-integrity file $H$ for writing, and subsequently attempts to read a low-integrity file $L$. If the process is downgraded at this point, we need to revoke its permissions to $H$. Applications typically assume that access permissions cannot be revoked, and hence may not handle the resulting access errors gracefully. On the other hand, if we deny the read access to $L$, it is likely to be better handled by the application.

To justify the third choice, consider an `SSH` server that reads low-integrity data from remote clients. The server code anticipates that clients may be malicious, and can reasonably be expected to protect itself adequately, thereby ensuring that the low-integrity input will not corrupt any high-integrity outputs. In contrast, the other two choices (deny or downgrade) will prevent the server from carrying out its function.

Analogous to the choices above, the following options are available when a low-integrity process attempts to write a high-integrity file (i.e., a file containing trustworthy data).

- **deny:** deny the access
- **downgrade:** downgrade the object to low-integrity, and allow the write operation
- **redirect:** redirect the access to a file $f$ so that it accesses another file $f_u$ instead. All subsequent accesses by an untrusted application to $f$ will be redirected to $f_u$, while accesses by a benign application won't be redirected.

To justify the second choice, consider a file that is created by copying a high-integrity file. By default, the copy would have a high-integrity label, but if the copy is subsequently used only in low-integrity applications, downgrading it is the best option, as it would permit this use. As a second example, consider a commonly executed shell command such as `cat x > y`. Here, the shell will first create the output file `y` before `cat` is launched. If the shell has high integrity, then `y` would be created with high integrity, but subsequently, if `x` turns out to have low integrity, the best option is to downgrade `y` rather than to deny access to `x`. On the other hand, if a file is known to be used by high-integrity applications, it should not be downgraded, and the write access must be denied.

To justify the third choice, consider a benign application that is sometimes used with untrusted objects, e.g., a word-processor that is used on a file from the Internet. During its run, this application may need to modify its preferences file, which would have a high-integrity label, since the application itself is benign. Denying this write operation can cause the word-processor to fail. Permitting the access would lower the integrity of the preferences file, leading to all future runs of the word-processor to have low integrity. However, by redirecting accesses to a low-integrity copy of the preferences file, both problems can be avoided.

In summary, there are several ways to resolve potential integrity violations (conflicts), and different resolutions are appropriate for different combinations of objects, subjects, and operations. Our (low-level) policy specifies, for each combination of object $O$ and subject $S$, which of the six different choices mentioned above should be applied. The sheer number of objects and subjects on modern operating systems (e.g., >100K files in system directories on typical Linux distributions) makes manual policy development a daunting task. We have therefore developed techniques to automate this task in the next section.

Although we discussed only file read/write operations above, the same concepts are applicable to other operations, e.g., file renames, directory operations, device accesses, and inter-process communication. We omit the details here, covering them briefly in the implementation section.

## 7.2 Automating Policy Development

The large number of objects and subjects in a modern OS distribution motivates automated policy development. We envision policy development to be undertaken by a security expert — for instance, a member of a Linux distribution development team. The goal of our analysis is to minimize the effort needed on the part of this expert. The input to the policy generation algorithm consists of:

- software package information, including package contents and dependencies,

- a list of untrusted packages and/or files,
- the list of integrity-critical objects,
- a log file that records resource accesses observed during normal operation on an unprotected system.

We use these to compute dependencies between objects and subjects in the system, based on which trust labels and policies are generated. *Ideally, all accesses observed in the log would be permitted by these policies* without generating user prompts or application failures, but in practice, some failures may be unavoidable. Naturally, it is more important to minimize failures of benign applications as opposed to untrusted ones. Given this goal of policy analysis, it becomes important to ensure coverage of all typical uses of most applications in the log, with particular emphasis on benign applications. Since usage is a function of users and their environments, better coverage can be obtained by analyzing multiple logs generated on different systems.

### 7.2.1 Computing Dependencies, Contexts and Labels

Critical to the success of our approach was the observation that software package information, such as those contained in RPM or Debian packages, can be leveraged for extracting subject/object dependencies. The package information indicates the contents (i.e., files) in each package, and the dependencies between different packages.

Since some of the dependences, such as those on configuration or data files, may not be present in the package database, we analyze the access log to extract additional dependency information. The dependencies can vary for the same application depending on the context in which it is used. For instance, when used by system processes (such as a boot-time script), `bash` typically needs to be able to write high-integrity files, and hence must itself operate at a high-level of integrity. However, when used by normal users, it can downgrade itself when necessary to operate on untrusted files. In the former context, `bash` reads and writes only high-integrity files, whereas in the latter context, it may read or write a mixture of high and low integrity files. To identify such differences, we treat each program $P$ as if it were several distinct programs, one corresponding to each of the following *execution contexts*: $P_s$ that is used by system processes, $P_a$ that is used by processes run by an administrator, and $P_{u_i}$ for processes run by a normal user $u_i$. (The distinction between $P_s$ and $P_a$ is that in the second case, the program is run by a descendant of an administrator's login shell.) For simplicity, we will assume that there is only one normal user $u$.

Once the logs are generated, it is straight-forward to compute the set of files read, written, or executed by a program in each of the above contexts. In addition, for each program and file read (written or executed) by that program, we compute the fraction of runs of that program in each context during which the file was read (written, or executed).

**Deriving Initial Object Labels.** Initial object labels are derived using the following steps. The assumption behind this algorithm is that all packages and files on the system are benign unless specified otherwise:

- Mark all packages that depend on untrusted packages as untrusted.
- Label all objects that belong to untrusted packages (or are explicitly identified as untrusted) with low integrity.
- Label any object that was written by an untrusted subject (i.e., a process whose executable or one of its libraries are from low-integrity files) with low integrity.
- Label all other objects as having high integrity.

We reiterate that object labels do not dictate policy: an object may have a high label initially, but the policy synthesized using the algorithm below may permit it to be written by an untrusted subject, which would cause its integrity label to become low.

### 7.2.2 Policy Generation

The policy generation algorithm consists of four phases as described below. The first phase identifies the set of "preserve-high" objects, i.e., objects whose integrity needs to be preserved. In the second phase, we generate the low-level policies for each (subject, object, access) combination, reflecting one of the policy choices described in Section 7.1.2. Phase III refines the policies by simulating the accesses contained in the log. Phase IV consists of ongoing policy refinement, as new applications are installed.

**Phase I: Identification of objects whose integrity needs to be preserved.**

1. *Initialize:*

    - Mark the initially specified integrity-critical objects as preserve-high.
    - For every program $P$ that is ever executed in the system context, mark $P$ as preserve-high.

2. *Propagate between package and object:*

    - For every object that is labeled as integrity-critical, the package that owns the object is marked as integrity-critical.
    - For every package $P$ such that an integrity-critical package depends on it, add $P$ to the set of integrity-critical packages.
    - For every object that belongs to an integrity-critical package (or object that is explicitly labeled as integrity-critical), mark it as "preserve-high."

3. *Propagate from object to subject.* If a program $P$ writes an object $Q$ that is marked preserve-high, then mark $P$ as preserve-high.

4. *Propagate from subject to object.* If a program $P$ reads or executes an object $Q$ then mark $Q$ as preserve-high if any of the following hold:

   (a) $P_s$ (i.e., $P$ operating in system context) reads/executes $Q$.

   (b) $P_a$ reads or executes object $Q$ in non-negligible fraction of runs, say, over 10% of the runs.

   (c) $P$ is marked preserve-high and $P_u$ almost always reads or executes $Q$, say, in over 95% of the runs.

   Every program that is run in system context is expected to be run in high-integrity mode, and hence the first rule. Most activities performed in administrator context have the potential to impact system integrity, and hence most of these activities should be performed in high integrity mode, and hence the second rule. For the third rule, if a benign program $P$ almost always reads or executes a specific file $Q$, then, if $Q$ has low integrity, it will prevent any use of $P$ in high integrity mode. It is unlikely that a benign program would be installed on a system in such a way that it is only executed at low-integrity level, and hence the third rule.

5. *Repeat the previous three steps until a fixpoint is reached.*

If any low-integrity file gets marked as preserve-high, there is a conflict and the policy generation cannot proceed until it is manually resolved. Such a conflict is indicative of an error in the input to the policy generation algorithm, e.g., a software package commonly used in system administration has been labeled as untrusted.

**Phase II: Resolution of conflicting accesses.** This phase identifies which of the policy choices discussed in Section 7.1.2 should be applied to each conflicting access involving (subject, object, access).

- **Deny policy:** For every object labeled preserve-high, the default policy is to permit reads by any subject but deny writes by low-integrity subjects. Similarly, for every object labeled with low-integrity, the default policy is to permit reads by low-integrity subjects but deny reads by high-integrity subjects. Exceptions to these defaults are made as described below, depending upon the program executed by a subject, and its trust level.

- **Downgrade subject policy:** A high-integrity subject $P$ running in context $c$ will be permitted to downgrade itself to low-integrity if there are runs in the log file where $P_c$ read a low integrity file, and did not write any high integrity objects subsequently. Such runs show that $P_c$ can run successfully, without experiencing security violations. If there are no such runs, then the downgrade policy is not used for $P_c$.

  Note that at runtime, a subject running $P_c$ may still be denied read access if it has already opened an object $O$ such that the policy associated with $O$ prevents its label from being downgraded.

  Finally, note that the use of context makes the downgrade policy more flexible. For instance, we may permit `bash` to downgrade itself when running in user mode, but not when it is run in system mode.

- **Trust policy:** Each subject $P$ that reads a low-integrity object and writes to an object marked preserve-high is a candidate for the "trust" policy. Such candidates are listed, and the policy developer needs to accept this choice. If this choice is not accepted, the log analyzer lists those operations from the log that would be disallowed as a result of this decision.

- **Downgrade object policy:** Any object that is not marked as preserve-high can be downgraded when it is overwritten by a low-integrity subject.

- **Redirect policy:** A redirect policy is applied to the $(P, O, write)$ combination if (a) $O$ is marked preserve-high, (b) $P$ reads $O$ in almost every run, and (c) $P$ writes $O$ in a non-negligible fraction of runs[1].

[1]These three conditions characterize the access by most applications to their preference files — the context in which the redirect policy was motivated.

**Phase III: Log simulation and policy refinement.** The algorithm described above did not take into account that file labels will change as the operations contained in the log file are performed. (If we did not make the simplifying assumption that the labels are static, then the analysis would become too complex due to mutual dependencies between policy choices and file labels.) To rectify this problem, we "simulate" the accesses found in the log. We track the integrity levels of objects and subjects during this simulation, and report all accesses that cause a violation of the policy generated in the previous step. The violation reports are aggregated based on the subject (or object), and are sorted in decreasing order of the number of occurrences, i.e., the report lists the subject (or object) with the highest number violations first. Subjects with high conflict counts are suggestive of programs that may need to be trusted, or untrusted programs that cannot be used.

Based on the conflict report, the policy developer may refine the set of trusted, benign, or untrusted programs. If so, the analysis is redone. In general, more than one iteration of refinement may be needed, although in our experience, one iteration has proven to be sufficient.

**Phase IV: Policy generation for new applications.** New files get created in the system. In addition, new applications may become available over time. In both cases, we cannot rely on any "logs" to generate policies for them. Our approach is as follows.

For objects that are created after policy deployment, their labels will be set to be the same as that of the subject that created them. The default policy for such newly created objects is that their labels can be downgraded when they are written by lower integrity subjects. In addition, accesses to these objects are logged. The resulting log is analyzed periodically using the same criteria described above to refine the initial policy. For instance, if the object is used repeatedly by high-integrity subjects, the policy would be set so that writes by lower-integrity subjects are denied.

If a new software package is installed, labels for the objects in the package are computed from the trust level of the package, which must be specified at the time of installation. The policies for these files are then refined over time, as the package is used by the user.

### 7.2.3 Soundness of Policies

Recall that the policies derived above are based on accesses observed on an unprotected system. Being unprotected, it is possible for the log to have been compromised due to malicious untrusted code. Thus, an important question is whether the soundness of the derived policies is compromised due to the use of such logs. An important feature of our policy generation technique is that this does not happen. Thus, if the generated policies are enforced on a newly installed system, these policies will preserve its integrity.

**Observation 2** *As long as the programs identified as* trusted *are indeed trustworthy, the policies generated above will preserve system integrity even if the access logs were compromised due to attacks.*

**Proof sketch:** Recall that preserving system integrity means that integrity-critical objects should never be written by low-integrity subjects. Observe that all integrity-critical objects are initialized to preserve-high in the first phase of the policy generation algorithm. The propagation steps in this phase can add to the set of objects marked preserve-high, but not remove any objects. In the next phase, note that "downgrade object" policy is applied only to those objects that aren't marked preserve-high. All other policy choices ensure that object labels will not be downgraded. Thus, the generated policy will ensure that the labels of integrity-critical objects remain high.

Observe that if the logs were compromised, far too many conflicts may be reported during policy generation. Worse, because the compromised logs may not reflect the behavior of programs on an uncompromised system, the generated policies may cause many accesses (not recorded in the log) to be denied, which can make the system unusable. Both these symptoms are suggestive of a compromised log file. The policy developer needs to obtain a new, uncompromised log and rerun the policy generation algorithm[2].

The above observation indicates that the primary weakness of *PPI* arises due to trusted programs. If they are incorrectly identified, or if they contain exploitable vulnerabilities, they can compromise end-user security objectives. This factor motivates features and techniques in the next section that limit and reduce the scope of trust.

## 7.3 Limiting Trust

Unlimited and unrestricted trust is often the weakest link in security, so we have incorporated features in *PPI* to reduce the scope and nature of trust placed on different programs. We describe these features below, followed by a discussion of how these features are used to address important applications such as software installers, browsers and email handlers, window systems, and so on.

**Invulnerable and Flow-Aware Applications.** All outputs of an *Invulnerable applications* continue to have high integrity even after reading low-integrity inputs. An example would be an ssh server that can be trusted to protect itself from potentially malicious network inputs, and maintain its high integrity.

*Flow-aware applications* can simultaneously handle inputs with different integrity levels. They keep track of which inputs affect which outputs, and label the outputs appropriately. (Our enforcement framework provides the primitives for flow-aware applications to control the labels on their outputs.) Flow-awareness is natural for some applications such as web-browsers that already keep track of the associations between their input actions and output

[2]Recall that end-users are not expected to generate policies, so they won't experience the security failures that result due to compromised logs; and hence we don't expect this possibility to negatively impact end-user experience.

actions. (Web browsers use this type of information to enforce the "same origin policy [22].") Alternatively, automated techniques such as runtime taint-tracking [41, 75, 49] may be used to achieve flow-awareness.

A generic technique to mitigate the risk due to excessive trust is to deploy defenses against the most common ways of exploiting applications using malicious inputs, e.g., address-space randomization [6, 74] or taint-tracking[3] [41, 75, 49]. This technique can be combined with a more specific risk mitigation mechanism described below that limits trust to certain contexts.

**Context-aware Trust.** A key observation is that programs are rarely designed to accept untrusted inputs on every input channel. For instance, while an ssh server may be robust against malicious data received over the network, it cannot protect itself from malicious configuration files, shared libraries or executables. Our approach, therefore, is to limit trust to the specific input contexts in which an application is believed to be capable of protecting itself. For an ssh server, this may be captured by explicitly stating that it is Invulnerable to inputs received on port 22.

With respect to files, one approach for limiting trust is to enumerate all the files read by an application in advance, and identify those that can have low integrity. This is far too cumbersome (or may not even be feasible) since the number of files read by an application may be very large (or unbounded). An alternative approach is to categorize a file as "data input" or "control input" (configuration or a library), and to permit a trusted application to read low-integrity data inputs but not control inputs. But manual identification of data and control inputs would be cumbersome. Instead, we rely on some of the properties of our policy synthesis techniques to achieve roughly the same effect. Specifically, note that configuration and library inputs will be read during practically every run of an application. As such, these files will be marked "preserve-high" in phase I of the policy generation algorithm, and hence the application will not be exposed to low-integrity configuration or

[3]Taint-tracking is preferable due to the weaknesses of ASR against local attacks.

library files[4].

### 7.3.1 Limiting Trust on Key Applications

**Software Installers** pose a particular challenge in the context of integrity protection. Previous techniques simply designated software installers as "trusted." This designation is problematic in the context of contemporary package management systems, where packages may contain arbitrary installation scripts that need to be run by the installer. During this run, they may need to modify files in system directories, and hence scripts cannot be run with low privileges.

We have developed a new approach to address this software installation problem. In our approach, the installer consists of two processes: a "worker" that runs as a low-integrity subject (but may have root privileges), and performs installation actions. To ensure that this low-integrity subject can overwrite system files if needed, a redirection policy is applied to all files written by this subject. A second high integrity "supervisor" subject runs after the first one completes. It verifies that the actions performed during installation were legitimate. In particular, it ensures that (a) the modifications made to the package management database are exactly those that were made to the file system, and (b) all the files installed are those that can be overwritten by a low-integrity subject. If the verification succeeds, the supervisor renames the redirected copies of files so that they replace their original versions. Otherwise, all the redirected copies are removed and the installation aborted.

**Web Browser and Email Handler.** Web browser and email client act as conduits for data received from the network. In our system, both web browser and email handler are considered flow-aware applications. Specifically, data

[4]This does not protect against the possibility that the application may, in a subsequent run, read a different configuration file. However, this is usually the result of running the application with a command-line option or an environment variable setting that causes it to read a different configuration/library file. These "inputs" are provided by a parent process, and hence are trusted, since the parent itself must have high-integrity in order for a child process to have high integrity.

received by a browser can be deemed high or low integrity based on the source of data and other factors such as the use of SSL. For the `Mozilla` browser used in our experiments, we built a small external program that uses the contents of a file "`downloads.rdf`" to correlate network inputs with the files written by the browser, and to label these files accordingly. We wrote a similar program for `pine` email reader.

**X-Server and Other Desktop Daemons.** GUI-based applications, called X-clients, need to access the X-Server. To ensure continued operation of benign as well as untrusted X-client applications, the X-Server should be made Invulnerable on input channels where they accept data from untrusted clients. We mitigate the risk due to this trust in two ways. First, X-server is made invulnerable only on inputs received via sockets used to connect to an untrusted client. Second, we make use of the X security extension [72] to restrict low-integrity applications so that they cannot perpetrate attacks on other windows that would otherwise be possible.

Unfortunately, due to the design of the GNOME desktop system, there are some servers (e.g., gconfd) that are used by multiple X-clients and need to be trusted in order to obtain a working system. We found that some of the recent results from the SE-Linux project [68, 26] could be applicable in this context.

**File Utilities.** Applications that can run at multiple trust levels can sometimes introduce some usability issues, specifically, when they are used to operate on input files with varying trust levels. We modified `cp` and `mv` to make them flow-aware, so that the output files correctly inherit the label from the input files.

## 7.4 Enforcement Framework

Our design is a hybrid system consisting of two components: a user-level library and a kernel-resident checker. Integrity-critical enforcement actions

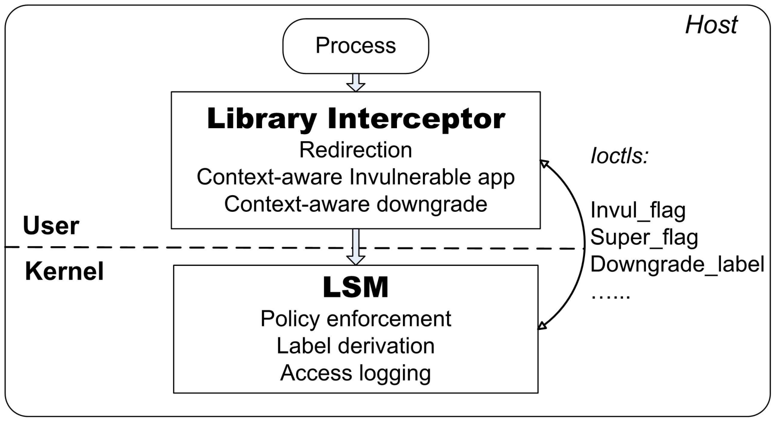


Figure 5: *PPI* System Architecture

are performed by the kernel component, while "functionality enhancement" features are relegated to the library. For instance, the kernel component does not deal with redirection policy. Moreover, while it supports the notion of trusted subjects, it does not concern itself with mechanisms for limiting trust, which are provided by the user-level component. While the kernel enforcement component is always trusted, the user-level component is trusted only when it operates in the context of a high-integrity process.

In our implementation, the kernel level component is realized using LSM (Linux Security Modules) [73], which has now become part of the standard Linux kernel. We use the security hooks of LSM to enforce information flow policies. Although our policy framework allows for policies to be a function of objects as well as subjects, for simplicity, the policies enforced by the kernel component have limited dependence on subjects. (More flexible subject-dependent policies can be realized using the user-level component.) This enables kernel-enforced policies to be stored with objects using *extended file attributes* available on major Linux file systems (including ext2, xfs, and reiserfs). Policies as well as integrity labels are stored using these attributes. Specifically, a 3-bit attribute *integ_obj* is used to store the integrity level of a file. (For extensibility, our implementation uses eight integrity levels.) A

11-bit policy is associated with each file, consisting of two parts. The first part pertains to read and write operations performed on the file:

- *down_obj* (3 bits) indicates the lowest integrity level to which this object can be downgraded.
- *log_obj* (1 bit) indicates whether accesses to this object should be logged. This feature could be used for auditing. In our implementation, it provides the mechanism for generating the logs used for policy generation.

The second part of the policy pertains to the use of a file to instantiate a subject, i.e., when the file is executed. It consists of the following components:

- *down_sub* (3 bits) indicates the lowest integrity level to which a process that executes this object can be downgraded.
- *log_sub* (2 bits) indicates whether accesses made by this subject should be logged. A second bit indicates whether this policy should be inherited by descendants of a subject.
- *invul_sub* (1 bit) indicates if this subject is invulnerable. No distinction is made among various subclasses of trusted applications described in Section 7.3 — it is up to the user-level component to implement distinctions such as flow-awareness and context-awareness.
- *super_sub* (1 bit) allows a subject to modify the labels associated with objects in the system. Naturally, this capability should be highly restricted. In our implementation, there is one administrative program that has this capability.

When *PPI* system is initialized, the extended attributes associated with all the files are populated using the labels and policies generated using the techniques in Section 7.2. New files inherit the integrity of the subject creating them. The log bits are set by default, super_sub and invul_sub bits are cleared, and the down_sub and down_obj bits are set to zero. (A lower integrity level or a higher downgrade level may be requested by any subject.)

After a fork, the child inherits the parent's attributes, including its integrity level. After an exec, the integrity of the subject is reduced (but can never be increased) to that of the file being executed. The super_sub policy is inherited from an executable file only if the subject is at the highest possible integrity level. Finally, the invul_sub as well as log_sub attributes are set from the executable file.

**Handling Devices, IPC, and File Renaming.** Devices typically need special treatment since they are not persistent in the same sense as files. The integrity and down_obj labels of devices such as `/dev/kmem`, `/dev/mem`, and `/dev/sda*` are set to be 111 to ensure that only the highest integrity subjects can modify them. Devices such as `/dev/null` and `/dev/zero` are treated as having low-integrity for writes, and high integrity for reads. The integrity labels of devices such as `/dev/tty*` and `/dev/pty/*` are derived from that of the associated login processes, in a manner similar to that of SELinux.

IPC and socket communication are treated as communication channels between subjects. As such, they inherit labels from the subjects that they connect. Moreover, since most of these communication mechanisms are bidirectional, subjects interconnected using them have identical security attributes[5]. The kernel enforcement component keeps track of "groups" of subjects interconnected using IPC and socket communication so that operations such as downgrading can be performed safely.

Finally, file renaming operations are deemed as a write operation on the file object.

**Implementing Subject Downgrading.** Subjects are downgraded at runtime only if doing so will not result in revocation of privileges. Specifically, the user-level component indicates, at the time of opening a file for read, whether a higher integrity subject is willing to downgrade to a lower level. The kernel

[5]An exception occurs in the case of trusted subjects that are invulnerable to inputs on a communication channel: in this case, the trusted process can continue to maintain its integrity level and other security attributes regardless of the security attributes of the subject on the other end of the communication channel.

component permits the downgrade if (a) the file can be successfully opened, (b) the subject does not have higher integrity files that it is writing into, and (c) all other subjects with which this subject has IPC can also be downgraded.

**User-Level Component.** The user-level component is implemented by intercepting system calls in `glibc`. Since the policies themselves are application-specific, their implementation is kept in a separate library. The user-level communicates with the kernel level using ioctl's to implement complex policies. We already described how "downgrade subject" policy is implemented through such coordination. It also supports *context-aware trust policies:* the user level determines whether a trusted application is Invulnerable (or flow-aware) in a certain context, and if so requests an open without downgrading its integrity. For flow-aware applications, the user-level communicates to the kernel level if files should be opened with a lower integrity level than that of the subject. The user level is also responsible for implementing the redirection policy. The kernel level is unaware of this policy, and simply treats it as a combination of a read operation on the original file, and a write operation on a new file.

# CHAPTER 8

# Evaluation of *PPI*

We initially implemented *PPI* on CentOS and subsequently migrated to an Ubuntu system. Some of our evaluations were performed on CentOS while others were performed on Ubuntu. The specifics of these two systems are as follows:

- CentOS 4.4 distribution of Linux with kernel version 2.6.9. The test machine has 693 rpm packages and 205k files in total.
- Ubuntu 7.10 distribution of Linux with kernel version 2.6.22-14. The test machine has 1164 dpkg packages and 159k files in total.

## 8.1 Experience

**Policy Generation on Ubuntu.** For generating policies, we used an access log collected over a period of 4 days on a personal laptop computer. We also carried out administrative tasks such as installing software, running backups, etc. The log file we obtained was around 1GB.

The set of initial integrity-critical file objects include files within `/boot`, `/etc/init.d/`, `/dev/sda`, and `/dev/kmem`. We identified 26 untrusted application packages, which include:

- *Media players:* mplayer and mpglen

- *Games:* gnome-games, crafty and angband
- *Instant messengers:* pidgin
- *Emulators:* apple2 and dosemu
- *File utilities:* rar, unrar and glimpse
- *X utilities:* axe
- *Java applications:* jta, sun-java6-bin and sun-java-jre

Based on the above initial configurations to log analysis, we performed the procedures described in Section 7.2 for label computation and policy generation.

Among all the files (159K) and dpkg packages (1164) on the system, the initial labels of 783 files (including files that belong to 26 untrusted packages and those written by them) were set to low integrity, while all the others are labeled high integrity initially.

Then we moved on to the policy generation phase. The number of subjects and objects that were assigned to different policy choices across different phases are summarized in Figure 2. In Phase I, the log analysis determined 73305 files (934 packages) that need to be marked "preserve-high." In Phase II, the analysis identified which of the six policy choices should be applied to each conflicting accesses. As a result, 168 programs that are exclusively run in system context were assigned "subject-deny" policy, and they won't be allowed to read untrusted input. 4 programs (Xorg, dbus-daemon, gconftool-2, and gnome-session) were marked "trusted," and they would retain high integrity level even when exposed to low-integrity input in certain channels, for instance, `/tmp/.X11-unix/X0` in the case of Xorg. All the rest of programs (6905) can be downgraded when reading low-integrity input. Correspondingly, for file objects, deny policies were applied to 73305 integrity critical objects, while the others can be downgraded. Finally, the analysis identified 15 file objects for redirection policy, including files such as the preferences files for gedit editor.

| | Phase I | | Phase II | | Phase III | |
|---|---|---|---|---|---|---|
| | subject | object | subject | object | subject | object |
| subject-deny | | | 168 | | 168 | |
| subject-downgrade | | | 6905 | | 6905 | |
| subject-trust | | | 4 | | 8 | |
| object-deny | | 73305 | | 73305 | | 73305 |
| object-downgrade | | 86185 | | 86185 | | 86185 |
| object-redirect | | | | 15 | | 16 |

Table 2: *PPI* Policy Generation in Different Phases.

Phase III used the initial policy configuration from Phase II. It reported 66 violations due to object and subject labels being downgraded in a running system. (A few thousand object and subject downgrades were observed.) Of all the violations, `trackerd` accounted for 51 conflicts. Another ten conflicts were due to `nautilus`, `gconfd-2`, and `bonono-activation-server`. After an analysis of the conflict report, it was determined that these four applications needed to be classified as trusted. Finally, four conflicts arose because `bash` could not write to the history file. This was resolved by using the redirect policy.

In addition, as described earlier, Mozilla and pine were also identified to be trusted applications. (One could of course avoid this for a browser by running two instances, one operating in high-integrity and used to access high-integrity web sites, and the other in low-integrity to access untrusted web sites.) Most of the trusted applications listed above should be made flow-aware, so they label their outputs appropriately, based on the integrity of their inputs.

***PPI* Experience during Runtime.** After we plugged in the policies generated in the previous step, we ran *PPI* in normal enforcing mode for several days. The system booted up without problems, which indicated that none of the init related programs were affected by low integrity input. We first ran all the normal administration type of tasks using root privilege, such as checking disk space, configuring network interfaces, etc. Then we logged in as normal user, and worked on the system as a typical user would. We also used all of the untrusted applications installed on the system. None of these activities raised any security violations. (We were able to create violations on purpose, e.g., by trying to edit a high and low integrity file at the same time using a benign editor.)

One slight problem is the side-effect of redirection policies: a duplicate copy of many of the preference files will be created as a result. One option is to periodically delete the low-integrity version of these files.

## 8.2 Effectiveness against Malicious Applications

Our integrity policy described in Section 7.2 provides effective defense against malware attacks.

- Linux rootkits. In this experiment, we downloaded around 10 up-to-date rootkits from [2]. Since our browser is made flow-aware, it checked the source of the downloaded software, and marked them as untrusted. User level rootkits such as bobkit, tuxkit, and lrk5 required an explicit installation phase. *PPI* reported permission violations such as deletion of /etc/rc.sysinit and /bin/ps, and hence their installation failed.

  Kernel level rootkits in the form of kernel modules are prevalent nowadays and are more difficult to detect. We downloaded, compiled, and installed one such rootkit, adore-ng. Since the initial download was low-integrity, the kernel module was also labeled with low-integrity. Since *PPI* does not permit loading of kernel modules with low integrity, this rootkit failed. Since only high integrity subjects are allowed to write to /dev/kmem, another kernel rootkit mood-nt failed with the error message ".D'ho! Impossible aprire kmem."

- Installation of *"Malicious" rpm package.* The Fedora package buildsystem [18] suggest three possible attack scenarios from the malicious package writer. Of these, a malicious rpm-scriptlet is a serious threat. To test the effectiveness of *PPI* under this threat, we crafted a "malicious" rpm package. This package is named `glibsys.rpm`. During the installation phase, the package tried to overwrite system files `/lib/libc.so` and `/bin/gcc`. These violations are captured by our system, and the installation aborted cleanly.

- Race condition attack. We crafted a piece of malware which employed a typical race condition attack. The attack we created is the classic TOCTTOU race condition [9], allowing a malicious process racing against a benign process in writing a high-integrity file (/etc/passwd) using a race condition by creating a symbolic link. This attack was successful on an unprotected system, but it was defeated by *PPI*, since the `follow_link` operation on the low-integrity symlink downgraded the benign process and the write was disallowed.

- Indirect attack. In this attack, we created another piece of malware that first created an executable with an enticing name and waited for users on the system to run it. The attack did not work as *PPI* automatically downgraded the process running the low integrity executable, and as a result, it could not overwrite any of the files in the system that can damage system integrity.

- Malformed data input. Similar to the above example, a malformed jpeg file was downloaded from some unknown source, so *PPI* marked it as low-integrity. When an image viewer opened it, although it was compromised, it was running in low-integrity mode, and hence its subsequent malicious actions failed.

## 8.3 Usability of *PPI* Versus Low Watermark

In order to better understand how *PPI* provides improved usability, we implemented a prototype version of Low watermark model using LSM, and applied the prototype to exactly the same host environment with the same initial

|  | read | write | stat | open/close | select | pipe latency |
|---|---|---|---|---|---|---|
| Orig | 2.1882 | 1.8670 | 7.9352 | 11.4859 | 27.2026 | 37.4196 |
| *PPI* | 2.3541 | 1.9899 | 15.0348 | 15.0348 | 28.5309 | 38.5261 |

Table 3: Microbechmark Result using LMbench. All numbers in microseconds.

labeling and policy configurations (including Invulnerable applications). We used the test environment for a period of one day and observed the violations in the following several types:

- Because of lack of object downgrade policy, in Low watermark model, gzip and ggv had difficulty in completing their jobs when handling low integrity files. In the case of gzip, it first ran with high integrity, created an output file, then downgraded on reading low-integrity input. Subsequently, gzip tried to change permissions on the output file, which was denied due to the fact that the file was at high integrity while the subject had been downgraded.

- With *PPI*, editors such as vi, gedit and gimp could be used to edit low-integrity as well as high-integrity files. With Low-watermark policy, the applications experienced a runtime error, if the first run of the application was performed with high-integrity input. In this case, the preference files were marked high-integrity. When the editor was subsequently used on a low-integrity file, it was downgraded, and its subsequent access to update the preference file was denied. If the first run was performed with low-integrity input, then the preference file was created with low integrity, which meant that every future run of the editor will be downgraded. In contrast, the use of log-based analysis in *PPI* enables these editors to work properly in both scenarios.

- As mentioned earlier, shell redirections typically cause problems due to self-revocation with Low-watermark model. For instance, when executing a command such as `cat in > out`, the shell, typically running at

| | Original | *PPI* Mode | |
|---|---|---|---|
| | Time | Time | Overhead |
| gzip | 1.269 | 1.271 | 0.1% |
| xpdf | 2.476 | 2.604 | 5.1% |
| make | 22.467 | 23.345 | 3.8% |

Table 4: Application Performance Overhead. All numbers are in seconds averaged across 10 runs.

high-integrity, creates the file `out` with high-integrity. If `in` is a low-integrity file, then `cat` will be downgraded on reading it, and then its attempt to write to `out` will be denied.

## 8.4 Performance Overheads

We present both the microbenchmark and macrobenchmark results for *PPI*. For microbenchmark evaluation, we used LMbench version 3 [38] to check the performance degradation of popular system calls. The results are summarized in Figure 3. We observed that *PPI* did not introduce noticeable overhead for most system calls except for open (and other similar system calls such as stat). For macrobenchmark, we measured 3 typical applications running within *PPI* during runtime. As illustrated in the Figure 4, the runtime overhead for applications in *PPI* is about 5% or less.

## 8.5 Limitations

Our approach cannot support arbitrary untrusted software. Some software, by its very nature, may need resource accesses that cannot safely be granted to untrusted applications. Our results show that for the type of programs that tend to be downloaded from untrusted sources, our approach is indeed applicable.

Our work does not focus on confidentiality or availability, but it still contributes to them in two ways. First, solutions for confidentiality and availability must build on solutions for integrity. Second, our techniques halt malware that exploits integrity violations to attack confidentiality; for example, by preventing a rootkit from installing itself, we also prevent it from subsequently harvesting and sending confidential account information. But no protection is provided from malware that targets violation of confidentiality without violating integrity.

# CHAPTER 9

# Related Work of *PPI*

**Information Flow Based Systems.** *Biba* model [8] has a strict "no read down and no write up" policy. The low watermark model [8] relaxes this strict policy to permit subjects to be downgraded when they read low-integrity inputs. *LOMAC* [19], a prototype implementation of low watermark model on Linux, addresses the "self-revocation" problem to a certain extent: a group of processes that share the same IPC can continue to communicate after one of the processes is downgraded by having the entire group downgraded, but the problem still remains for files. *SLIM (Simple Linux Integrity Model)* [53] is part of the IBM research trusted client project, and is also based on the LOMAC model. It also incorporates the Caernarvon model [25], which supports verified trusted programs and limits the trust on these programs by separating read and execute privileges. The features developed in this part are more general in this regard, allowing distinctions between data input and control input, and so on.

IX [37] is an experimental MLS variant of Unix. It uses dynamic labels on processes and files to track information flow for providing privacy and integrity. In contrast, our technique generalizes the LOMAC model by offering several other policy choices, which we have shown to be helpful for improving usability. Other important distinctions between these works and ours are that we decouple policies from labels, and provide automated techniques for policy development.

*Clark-Wilson* Model [14] is different from the above multilevel security models, it emphasizes "well-formed transaction" and "separation of duty". It defines a set of certification and enforcement rules that guarantee the integrity for constraint data items. And it requires verification of applications (which is not possible for current technologies) and verification of information integrity properties of the inputs, which is another difficult task. Also, this model is not suitable for environment such as operating systems, for which the operations are not "well-formed transactions". Clark-Wilson Model defined that untrusted processes should not send unfiltered inputs to trusted processes and [60] provided an automatic way for verification. This is similar as the way we achieve the trust limit on trusted programs. The difference is that it used source code annotation, while our approach uses runtime mechanism.

*Windows Vista* enforces only the "no write up" part of an information flow policy, "no read down" is not enforced as it causes usability problems. Unfortunately, malware writers can adapt their malware to defeat this protection, as discussed in the introduction. In contrast, *Back to the Future* system [21] enforces only the "no read down" policy. Its main advantages are that it can recognize any attempt by malware to inject itself into inputs consumed by benign applications, and the ability to rollback malware effects. A drawback is that any attempt to "use" the output of an untrusted (but not malicious) application would require user intervention. It can be difficult for users to judge whether such inputs are "safe" and respond correctly to such prompts. Secondly, malware can freely overwrite critical files, which need to be "recovered" when the data is subsequently accessed — a potentially time-consuming operation.

Virtual machines [69, 16, 5] rely on isolation to confine untrusted processes. While isolation is an effective protection technique, maintaining multiple isolated working environments is not very convenient for users. In particular, objects such as files that need to be accessed by untrusted code have to be copied into and/or out of the isolated environment each time.

Li et al [31] also address the problem of making mandatory access control usable by focusing on techniques for policy development. However, their focus is on servers exposed to network attacks, as opposed to untrusted software threats on workstations. The nature of the threat (remote attacks versus adaptive malware) is quite different, causing them to focus on techniques that are quite different from ours. For instance, they don't protect user files, while we consider corrupting of user files to be a very powerful attack vector in our context. Also, the normal execution of most untrusted software would fail when disallowing their write to files that are not world-writable. Moreover, they do not consider the problem of securing software installations, or provide analysis techniques that can leverage resource access logs to generate policies. Nevertheless, there are some similarities as well: we have been able to use their notion of limiting trust to certain network channels. In addition, we provide a refinement of this notion in the context of files.

All of the above works were based on centralized policies, which are less flexible than decentralized information-flow control (DIFC) policies. DIFC policies allow applications to control how their data is used. In this regard, JFlow [40] is a language level approach. Asbestos [17] and Hi-Star [78] are new operating system projects that have been architected with information flow mechanisms incorporated in their design. Flume [29] is focused on implementing an extension to existing operating systems to provide process level DIFC. Like most other previous works in information-flow based security, these projects too focus on mechanisms, whereas our focus has been on generating the policies needed to build working systems.

**SELinux, Sandboxing and Related Techniques.** Several techniques have been developed for sand-boxing [20, 4, 48]. Model-carrying code [59] is focused on the problem of policy development, and provides a framework for code producers and code consumers to collaborate for developing policies that satisfy their security goals. Nevertheless, development of sandboxing policies that can robustly defend against adaptive malware is a challenge due to the ease of indirect attacks as described in the Introduction.

AppArmor [1] and Smack [55] are MAC protection mechanisms on Linux. Different from *PPI*, they mainly focus on mechanisms instead of policies, and they do not track information flow between benign and untrusted applications, hence suffer from indirect attacks. TOMOYO [3] is another MAC Linux protection project. It uses program execution chain as domain, and enforces access policies learned during runtime before hand by the end user. Without explicitly specifying the trustworthiness of the programs on the system, the "learned" accesses from malicious programs might be allowed and enforced during runtime. It is unreasonable to assume that normal users can figure out what accesses to allow or disallow.

SELinux [35] uses domain and type enforcement (DTE) policies to confine programs. Their main focus has been on servers, and they have developed very detailed policies aimed at providing the least privilege needed by such applications. Systrace project [48] has also developed system-call based sandboxing policies for several applications, and is widely used in FreeBSD. Neither approach ensures system integrity by design. SELinux as well as Systrace can log accesses made during trial runs of an application, and use it as the basis to generate a policy for that application. Their policy generation technique such as [61] is useful for benign code, such as servers, but would be dangerous for untrusted applications.

Whereas our focus is on generating policies that ensure integrity, other researchers have worked on the complementary problem of determining whether a given policy guarantees system integrity [23].

# CHAPTER 10

# Conclusion

As malware attacks become more and more sophisticated, previous techniques are either lack of complete protection or are impractical to use because of usability problems and the difficulty in policy development. This dissertation presented techniques for providing integrity protection on contemporary operating system distributions: on the one hand, our approach provides positive assurances against malware from damaging system integrity; on the other hand, the approach can be readily applied on contemporary OSes with minimal usability problems.

The *SEE* approach is especially suitable for end users to run stand-alone untrusted applications by isolating their effects from the main system. One-way isolation is used to achieve both light-weight environment duplication and effect isolation. It is versatile enough to support a wide range of other applications. A key benefit of this approach is that it provides strong consistency. In particular, if the results of isolated execution are not acceptable to a user, then the resulting system state is as if the execution never took place. On the other hand, if the results are accepted, then the user is guaranteed that the effect of isolated execution will be identical to that of atomically executing the same program at the point of commit.

The *PPI* approach is more general and suitable for the whole system integrity protection. One of the central problems in developing practical systems based on mandatory access control policies has been the complexity of policy development. We have developed techniques to automate the generation of low level information flow policies from data contained in software package managers, and access logs that capture normal usage of these systems. Our experimental results show that the technique is efficient, it can provide robust protection from most malware, and does not pose significant usability problems.

Although malware attacks against computer integrity can be deployed in a diversified way, they basically follow the same principle, that is, the attacks try to utilize unacceptable information flows from them to integrity-critical system resources. The research presented in this dissertation provides new directions in incorporating practical information flow based mechanisms to thwart malware attacks in contemporary operating systems. *SEE* can be deemed as a quick solution for end users to run untrusted software. *PPI* offers a more systematic protection mechanism, the policies can be synthesized at the OS distribution site and then deployed to end user systems so that thousands (if not millions) of users can benefit from the integrity protection automatically.

# Bibliography

[1] Apparmor linux application security. http://www.novell.com/linux/security/apparmor/.

[2] Linux rootkits. http://www.eviltime.com/download.php?page=hacking&subpage=rootkits.

[3] Tomoyo linux project. http://tomoyo.sourceforge.jp/index.html.en.

[4] A. Acharya and M. Raje. Mapbox: Using parameterized behavior classes to confine applications. In *USENIX Security Symposium*, 2000.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, volume 37, 5 of *Operating Systems Review*, pages 164–177, New York, Oct. 19–22 2003.

[6] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficent approach to combat a broad range of memory error exploits. In *Proceedings of the 12th Usenix Security Symposium*, Washington, D.C., August 2003.

[7] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, Mitre Corporation, June 1975.

[8] K. J. Biba. Integrity considerations for secure computer systems. In *Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts*, 1977.

[9] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2), 1996.

[10] A. Brown and D. Patterson. Undo for operators: Building an undoable e-mail store. In *USENIX Annual Technical Conference*, 2003.

[11] P. M. Chen and B. D. Nobl. When virtual is better than real. In *Proceedings of Workshop on Hot Topics in Operating Systems*, 2001.

[12] T. Chiueh, H. Sankaran, and A. Neogi. Spout: A transparent distributed execution engine for java applets. In *Proceedings of International Conference on Distributed Computing Systems*, 2000.

[13] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode file system. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 43–60, San Fransisco, CA, USA, 1992.

[14] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *IEEE Symposium of Security and Privacy*, pages 184–194, 1987.

[15] A. Dan, A. Mohindra, R. Ramaswami, and D. Sitaram. Chakravyuha: A sandbox operating system for the controlled execution of alien code. Technical report, IBM T.J. Watson research center, 1997.

[16] J. Dike. A User-Mode port of the linux kernel. In *Proceedings of the 4th Annual Showcase and Conference (LINUX-00)*, pages 63–72, Berkeley, CA, Oct. 10–14 2000.

[17] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazires, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *20th Symposium on Operating Systems Principles (SOSP 2005)*, 2005.

[18] *The fedora.us buildsystem.* http://enrico-scholz.de/fedora.us-build/html/.

[19] T. Fraser. Lomac: Low water-mark integrity protection for COTS environments. In *IEEE Symposium on Security and Privacy*, 2000.

[20] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications: confining the wily hacker. In *USENIX Security Symposium*, 1996.

[21] F. Hsu, T. Ristenpart, and H. Chen. Back to the future: A framework for automatic malware removal and system repair. In *Annual Computer Security Applications Conference (ACSAC)*, December 2006.

[22] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 737–744, New York, NY, USA, 2006.

[23] T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the selinux example policy. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[24] S. Jajodia, P. Liu, and C. D. McCollum. Application-level isolation to cope with malicious database users. In *Annual Computer Security Applications Conference (ACSAC)*, 1998.

[25] P. Karger, V. Austel, and D. Toll. Using a mandatory secrecy and integrity policy on smart cards and mobile devices. In *EUROSMART Security Conference*, pages 134–148, Marseilles, France, 2000.

[26] P. Karger, V. Austel, and D. Toll. Using gconf as an example of how to create an userspace object manager. In *SELinux Symposium*, 2007.

[27] J. Katcher. Postmark: A new file system benchmark. Technical Report TR3022, Network Applicance Inc., 1997.

[28] D. G. Korn and E. Krell. A new dimension for the unix file system. *Software: Practice & Experience*, 20(S1), 1990.

[29] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 321–334, New York, NY, USA, 2007.

[30] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[31] N. Li, Z. Mao, and H. Chen. Usable mandatory integrity protection for operating systems. In *IEEE Symposium on Security and Privacy*, 2007.

[32] Z. Liang, V. Venkatakrishnan, and R. Sekar. Isolated program execution: An application transparent approach for executing untrusted programs. In *Proceedings of Annual Computer Security Applications Conference (ACSAC)*, 2003.

[33] P. Liu, S. Jajodia, and C. D. McCollum. Intrusion confinement by isolation in information systems. In *Proceedings of IFIP Workshop on Database Security*, 1999.

[34] Loop back file system. Unix man page.

[35] P. A. Loscocco and S. D. Smalley. Meeting critical security objectives with security-enhanced linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, 2001.

[36] D. Malkhi and M. K. Reiter. Secure execution of java applets using a remote playground. *Software Engineering*, 26(12), 2000.

[37] M. D. McIlroy and J. A. Reeds. Multilevel security in the UNIX tradition. *Software - Practice and Experience*, 22(8):673–694, 1992.

[38] L. McVoy and C. Staelin. Lmbench. http://www.bitmover.com/lmbench/.

[39] K.-K. Muniswamy-Reddy, C. P. Wright, A. P. Himmer, and E. Zadok. A versatile and user-oriented versioning file system. In *Proceedings of USENIX Conference on File and Storage Technologies*, 2004.

[40] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[41] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of 12th Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[42] J.-S. Pendry and M. K. McKusick. Union mounts in 4.4bsd-lite. In *USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems (New Orleans)*, pages 25–33, 1995.

[43] J. S. Pendry, N. Williams, and E. Zadok. Am-utils user manual, 6.1b3 edition, july 2003. http://www.am-utils.org.

[44] Z. Peterson and R. Burns. Ext3cow: The design, implementation, and analysis of metadata for a time-shifting file system. Technical Report HSSL-2003-03, Hopkins Storage Systems Lab, Department of Computer Science, Johns Hopkins University, 2003.

[45] Picturepages software. Distributed on the Internet. http://www.canonical.org/picturepages/.

[46] D. Pilania and T. cker Chiueh. Design, implementation, and evaluation of an intrusion resilient database system. In *International Conference on Dependable Systems and Networks*, 2003.

[47] V. Prevelakis and D. Spinellis. Sandboxing applications. In *Proceedings of Usenix Annual Technical Conference: FREENIX Track*, 2001.

[48] N. Provos. Improving host security with system call policies. In *Proceedings of the 11th USENIX Security Symposium*, pages 257–272, 2003.

[49] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting general

security attacks. In *IEEE/ACM International Symposium on Microarchitecture*, December 2006.

[50] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of USENIX Conference on File and Storage Technologies*, Monterey, CA, 2002.

[51] Recovery-oriented computing. http://roc.cs.berkeley.edu.

[52] W. D. Roome. 3dfs: A time-oriented file server. In *Proceedings of the USENIX Winter 1992 Technical Conference*, 1991.

[53] D. Safford and M. Zohar. A trusted linux client (tlc), 2005.

[54] D. J. Santry, M. J. Feeley, N. C. Hutchinson, and A. C. Veitch. Elephant: The file system that never forgets. In *Proceedings of Workshop on Hot Topics in Operating Systems*, 1999.

[55] C. Schaufler. The simplified mandatory access control kernel. http://www.schaufler-ca.com/data/SmackWhitePaper.pdf.

[56] K. Scott and J. Davidson. Safe virtual execution using software dynamic translation. In *Proceedings of Annual Computer Security Applications Conference*, 2002.

[57] R. Sekar, Y. Cai, and M. Segal. A specification-based approach for building survivable systems. In *National Information Systems Security Conference*, Oct 1998.

[58] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *Proceedings of the USENIX Security Symposium*, 1999.

[59] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. In *ACM Symposium on Operating System Principles*, Bolton Landing, New York, October 2003.

[60] U. Shankar, T. Jaeger, and R. Sailer. Toward automated information-flow integrity verification for security-critical applications. In *Network and Distributed Systems Security Symposium*, 2006.

[61] B. Sniffen, J. Ramsdell, and D. Harris. Guided policy generation for application authors. In *SELinux Symposium*, 2006.

[62] C. Soules, G. Goodson, J. Strunk, and G. Ganger. Metadata efficiency in a comprehensive versioning file system. In *Proceedings of USENIX Conference on File and Storage Technologies*, 2002.

[63] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1):77–110, Feb. 2005.

[64] Translucent file system, 1990. SunOS Reference Manual, Sun Microsystems.

[65] T. Tiilikainen. Rename-them-all, linux freeware version. http://linux.iconet.com.br/system/preview/8622.html.

[66] P. Uppuluri. *Intrusion Detection/Prevention Using Behavior Specifications*. PhD thesis, Stony Brook University, 2003.

[67] Vmware. URL. http://www.vmware.com.

[68] E. F. Walsh. Integrating xfree86 with security-enhanced linux. In *X Developers Conference*, Cambridge, MA, 2004.

[69] B. Walters. VMware virtual platform. *j-LINUX-J*, 63, July 1999.

[70] Webstone, the benchmark for web servers. http://www.mindcraft.com/webstone.

[71] A. Whitaker, M. Shaw, and S. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *USENIX Annual Technical Conference*, 2002.

[72] D. P. Wiggins. Security extension specification, version 7.0. Technical report, X Consortium, Inc., 1996.

[73] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, USA, August 5-9, 2002*, pages 17–31. USENIX, 2002.

[74] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Symposium on Reliable and Distributed Systems (SRDS)*, Florence, Italy, October 2003.

[75] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, August 2006.

[76] E. Zadok, I. Badulescu, and A. Shender. Extending file systems using stackable templates. In *Proceedings of USENIX Annual Technical Conference*, 1999.

[77] M. Zalewski. Fakebust, a malicious code analyzer, 2004. http://www.derkeiler.com/Mailing-Lists/securityfocus/bugtraq/2004-09/0251.html.

[78] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazires. Making information flow explicit in histar. In *Seventh Symposium on Operating Systems Design and Implementation (OSDI06)*, 2006.

[79] N. Zhu. Data versioning systems. Technical report, Stony Brook University, http://www.ecsl.cs.sunysb.edu/tech_reports.html.

[80] N. Zhu and T. Chiueh. Design, implementation, and evaluation of repairable file service. In *Proceedings of International Conference on Dependable Systems and Networks*, 2003.