Enhancing Operating Systems with Network Provenance Based Policies for Systematic Malware Defense

A Dissertation presented

by

Wai Kit Sze

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

May 2016

Stony Brook University

The Graduate School

Wai Kit Sze

We, the dissertation committee for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation

Dr. R. Sekar - Dissertation Advisor
Professor of Computer Science

Dr. Donald Porter - Chairperson of Defense
Assistant Professor of Computer Science

Dr. Long Lu - Committee member of Defense
Assistant Professor of Computer Science

Dr. Trent Jaeger- Committee member of Defense
Professor of Computer Science and Engineering Department,
Pennsylvania State University

This dissertation is accepted by the Graduate School

Charles Taber
Dean of the Graduate School

Abstract of the Dissertation

Enhancing Operating Systems with Network Provenance Based Policies for Systematic Malware Defense

by

Wai Kit Sze

Doctor of Philosophy

in

Computer Science

Stony Brook University

2016

Todays OSes adopt users as the basic unit of trust. Every file and process owned by the same user has the same userid as the user. This design stems from the very first multi-user OS created, a time when computers were self-contained, and file contents were under the control of users. Today, users frequently download data and code from the Internet, without fully understanding their content or consequences. However, existing desktop OSes reuse the same old trust model and treat downloaded files as if users are fully responsible for them. This trust is exploited by today's malware.

In this dissertation, we generalize the existing OS trust hierarchy with remote provenance information. Instead of having only mutually-untrusted users, we extend it to principals encoding both local user and remote provenance information. We allow principals to have arbitrary trust relationships. With just two provenances having a unidirectional trust relationship, we can already build a usable integrity protection that can systematically defend against unknown malware. In addition, we show how our framework substantially generalizes previous ones such as the web browsers' same-origin policy and the policies governing inter-app interactions on mobile OSes.

Trust hierarchy and access controls are enforced deep inside OSes. Generalizing the trust model can affect all applications and every component in OSes. Instead of building a new OS from scratch or instrumenting existing OSes to enforce this new trust model directly, we re-purpose existing security mechanism common in contemporary OSes to achieve this generalization. This re-purposing mediates every access automatically, incurs low performance overhead, and is agnostic to both OSes and applications. Our system has been implemented on Linux, BSD, and Windows, supporting large applications like Firefox, Microsoft Office, Adobe Reader and Photoshop.

This dissertation is organized into three parts. The first part is concerned with provenance tracking and enforcement mechanisms. Our main contributions in this part are (a) a novel dual-sandbox architecture that provides strong security against untrusted (potentially malicious) code, while preserving compatibility with the vast base of existing applications, and (b) an approach for encoding provenance using userids supported on contemporary operating systems, which enables the enforcement framework to be easily implemented on Linux, BSD and Windows.

The second part of the dissertation studies provenance-based security policies. Our key contributions in this context include: (a) a formal treatment of the usability versus functionality trade-off made by various integrity-preservation policies, (b) the development of a new integrity policy that, in a formal sense, provides an optimal trade-off, (c) formalizing what it means for a policy to preserve the integrity and availability, and establishing that our policies indeed achieve these goals, (d) development of inference techniques to automate several components of policy development, and (e) the development of a general provenance-based security policy framework that is shown to subsume existing models such as those arising in the context of web mashups and smart phone apps.

The third part of this dissertation implements the mechanisms and policies developed in the previous parts into several prototype systems and evaluates their effectiveness, performance and usability. The first system, SPIF, is an integrity protection system for commodity OSes, including Linux, BSD, and Windows. SPIF can run large, unmodified applications, such as Firefox, Google Chrome, Microsoft Office, Adobe Reader, and Photoshop, without any impact on user experience, while warding off sophisticated and stealthy malware. The second system, SRFD, addresses a long-standing problem in information flow tracking, called self-revocation. The last system, SWINST, is a system to secure the software installation process. We use SWINST to demonstrate the need for rollback and commit capabilities in an enforcement mechanism, and how these can be utilized to realize highly expressive security policies that cannot be supported otherwise. This system has been successfully evaluated on over 20,000 software packages available on Ubuntu Linux.

Contents

List of Figures

Acknowledgements

Though only my name appears on the cover of this dissertation, a great many people have contributed to its production. I owe my gratitude to all those people who have made this dissertation possible and because of whom my graduate experience has been one that I will cherish forever. My deepest gratitude is to my advisor, Dr. R. Sekar. I have been amazingly fortunate to have an advisor who gave me the freedom to explore on my own, and at the same time the guidance to recover when my steps faltered. Sekar taught me how to question thoughts and express ideas. His patience and support helped me overcome many crisis situations and finish this dissertation.

Dr. Donald Porter's insightful comments and constructive criticisms at different stages of my research were thought-provoking, and they helped me focus my ideas and shape my research. I am grateful to him for providing valuable feedbacks. I am also thankful to him for encouraging the use of correct grammar and scientific writing style in my writings and for carefully reading and commenting on this manuscript.

I am grateful to Dr. Trent Jaeger for his practical advice. His suggestions and questions helped me understanding and enriching my ideas.

I am also indebted to the members of the AT&T Labs— Research with whom I have interacted during the course of my graduate studies. Particularly, I would like to acknowledge Dr. Abhinav Srivastava for the many valuable discussions that helped me understand how to research in security.

I am also grateful to the following staff at Stony Brook University for their various forms of support during my graduate study– Cynthia SantaCruz-Scalzo, Betty Knittweis, Kathy Germana, Stella Mannino, and Brian Tria.

Many friends have helped me stay sane through these difficult years. Their support and care helped me overcome setbacks and stay focused on my graduate study. I greatly value their friendship, and I deeply appreciate their belief in me. In particular, I would like to thank my lab mates from the Secure System Lab at Stony Brook University. I really enjoy discussing problems with them, including Alireza Saberi, Niranjan Hasabnis, Riccardo Pelizzi, Rui Qiao, Mingwei Zhang, Tung Tran, Laszlo Szekeres, and Md Nahid Hossain. I am also grateful to the Chapin community that helped me adjust to a new country.

Most importantly, none of this would have been possible without the love and patience of my family. My immediate family to whom this dissertation is dedicated to, has been a constant source of love, concern, support and strength all these years. I would like to express my heart-felt gratitude to my family. My extended family has aided and encouraged me throughout this endeavor.

Finally, I appreciate the financial support from NSF (CNS-0831298, CNS-1319137), AFOSR (FA9550-09-1-0539, and DARPA (contract FA8650-15-C-7561) that funded parts of the research discussed in this dissertation.

Chapter 1

Introduction

The earliest computers were mainframes from the 1950s that lacked any form of operating system. Each user had sole use of the machine for a scheduled period of time. They would arrive at the computer with program and data, often on punched paper cards. The program would be loaded to the machine, and the machine would be set to work until the program completed or crashed.

As computers became faster, people quickly realized that rapid advances in electronics could allow time-sharing of hardware resources across multiple users. The very first multi-user OS, called Multics, was then created in the late 1960s, allowing multiple users to use a computer simultaneously. Users became the basic unit of isolation to prevent one user's task from affecting that of another. Every resource in a multi-user OS is labeled based on users. This notion made sense since computers were self-contained and isolated systems: users provide their own programs and data in the form of punched cards or magnetic taps. There was no way to introduce new data or code into a computer (except for a dedicated system administrator that could access some backup and restore devices). The only way data or code could be on the system is if the user actually produced or generated it by running existing programs on some of his/her data. Hence, the user is solely responsible for the data.

When the Internet became popular in the 1980s, no changes were made to the user/permission model to account for the possibility that code or data could be downloaded from the Internet. The way the implementation handled this possibility was to set the ownership of a downloaded file to that of the process performing the transfer. This labeling only captures the fact that the download was handled by the process, but it fails to meaningfully account for the network origin of the file content.

As sharing across the Internet became the norm in the 1990s, people share not only data but also code, a.k.a. third-party applications. Many users do not hesitate to use resources that are readily available from the Internet without fully considering the security implications. Unfortunately, traditional OS-based mechanisms provide no protection in these cases, since OSes possess no information about the true source of such files.

Traditional OS-based mechanisms are unprepared for the growing use of software from untrusted sources. The problem is that they do not take software source into account, but make all security decisions on the basis of the user, who does not have much control over the operation of software. As a result, when a process performs operations such as adding a file to run during system booting, OSes cannot distinguish if the user intentionally added this file, or it was an unintended side-effect of running an application from the Internet.

Newer environments such as mobile OSes and web browsers provide better security against code and data downloaded from Internet sites. The basic idea is to isolate code from different sources using the Same-Origin Policy (SOP) or App model, so that code from different sources cannot interact with each other. If a domain or an app is malicious or compromised, it can only affects its own data, but is unable to attack or subvert code/data from other sites. This contrasts with desktop OSes, where a single malicious application is often capable of corrupting all users and/or system data/code on the system.

While isolation can bring increased security, it prevents applications from collaborating with each other. Familiar application composition constructs such as pipelines, shared libraries, plugins, IPCs, and scripts can no longer be used if they involve code or data from different sites. Browsers and mobile OSes have hence introduced newer constructs such as the post-message API [Mozilla Developer Network, 2015] on browsers and the Intent mechanism on Android [Google, 2016] in order to support limited form of collaboration for code/data from different origins. Unfortunately, these mechanisms are much more limited than the composition mechanisms on desktop OSes. Worse, to the extend collaboration takes place, it becomes possible for one of the sites to compromise the integrity of another.

To protect desktop systems, we need a new design that is compatible with existing applications and support all of the composition mechanisms available on today's OSes, while simultaneously providing strong security guarantees. Ideally, the design should:

- *Allow code from different sources to cooperate.* The hallmark of today's OSes is the ability to compose applications together. UNIX pipelines represented one of the early examples of application composition. Other common forms of composition can happen through files or scripts, e.g., printing a spread sheet into a PDF file and then emailing this PDF file. We need a new design that enables data sharing across applications, while supporting mechanisms and tools such as pipelines, scripts, library, and plugins.

- *Provide strong security guarantees without needing to make optimistic assumptions or the need to place unlimited trust on the platform provider.* Regardless of how careful a piece of code is written, attackers can still find vulnerabilities in it. We need a design that embraces the fact that most code is vulnerable.

- *Preserve compatibility with existing OSes and applications.* Ideally, the design does not require any change to existing OSes and applications so that the design is readily deployable. The design should not require efforts to develop security policies for each application.

- *Provide a framework that generalizes known defenses used on browsers, mobile OSes, and desktop OSes.*

1.1 Overview and Contributions

This dissertation explores the use of provenance, i.e., information origins of resources, to enhance system security in contemporary desktop OSes. Newer environments already leverage provenance information to enforce isolation policy based on provenance of the code. While isolation is effective in confining malicious applications from compromising others, isolation also restricts how applications can interact. A natural question is *"Can we use the provenance information to achieve better security while maintaining compatibility with existing OSes?"*. To answer this question, we need three parts: enforcement mechanisms, policies, and systems. Enforcement mechanisms provide the basis for enforcing policies. Policies dictate the security, functionality and usability of the systems. Finally, an overall system implementation and experimental evaluation can illustrate the use of provenance.

1.1.1 Enforcement mechanisms

In Part I, we first focus on the enforcement mechanisms for realizing provenance tracking policies. The two main fundamental features of provenance tracking are (1) how to label subjects and objects, and (2) how to enforce policies when subjects and objects interact.

The most natural approach is to modify OSes to handle provenance labels on subjects and objects directly. This approach needs kernel modifications. The advantage of implementing mechanisms inside kernel is that these mechanisms have good security against malware. They can even protect malware with administrative privileges. However, they are closely tied to specific OSes.

A more interesting question is whether provenance tracking is possible at the user-level. We propose a novel dual-sandbox architecture, which confines not only untrusted processes, but also benign processes, to realize user-level provenance tracking. Instead of storing provenance information using custom labels, our design overloads existing permission (Discretionary Access Control) labels to encode provenance information. The dual-sandbox architecture (Chapter 4) allows us to provide strong security against untrusted (potentially malicious code) while preserving compatibility with the vast base of existing applications. Since the system is designed entirely in user-space without requiring any kernel modification, we have been able to port the system to different OSes easily, including Ubuntu, PC-BSD, and Windows.

1.1.2 Policies

After discussing the enforcement mechanism aspect for realizing provenance tracking, we discuss in Part II the policy aspect.

To make the discussion more concrete, we first discuss policies specifically in the context of preserving system integrity in Chapter 5. We propose a formal treatment of the usability versus functionality trade-off made by various integrity-preservation policies. We show that existing policies such as Biba and low-water-mark [Biba, 1977] focus on either usability or functionality, and therefore they are not optimal. We develop a new integrity policy called

SRFD that, in a formal sense, provides an optimal trade-off. We also develop a policy called ED/UII that is optimal in most common use cases.

We also formalize what it means for a policy to preserve the integrity and availability, and establishing that our policies indeed achieve these goals. We develop inference techniques in Chapter 6 to automate several components of policy development. For example, if an operation to modify a high-integrity file is going to be denied, how a policy denies the operation (e.g., by returning what error code) could affect program behaviors. While a policy developer can specify policy for each file and each operation, this is a troublesome task. Our technique relies on inferring file type based on access behaviors, and applies different policies based on the inferred type.

Finally, we develop a general provenance-based security policy framework in Chapter 7 that is shown to subsume existing models such as those arising in the context of web mashups and smart phone apps.

1.1.3 Systems

In Part III, we present a number of systems that implement the enforcement mechanisms and policies proposed in the previous chapters. We discuss implementation and evaluation for each the system:

- **Spif: Integrity protection system for commodity OSes**.
  We present a system called Secure Provenance-based Integrity Fortification (Spif), which combines our dual-sandboxing architecture with ED/UII policy for malware defense. We have implemented Spif on multiple platforms, including Linux, BSD, and Windows. Spif also supports running large, unmodified applications such as Microsoft Office, Internet Explorer, Firefox, Chrome, Windows Media Player, Adobe Reader, and Photoshop. We discuss in detail how we implement the dual-sandbox architecture without modifying OSes. We also discuss the security guarantees that Spif provides. We present an extensive performance and security evaluation of Spif. We extend Spif to enforce the generalized policy presented.

- **Integrity protection using SRFD with dynamic downgrading**.
  SRFD is our kernel based implementation of the Self-Revocation Free Downgrading policy for malware defense. We implemented SRFD on Ubuntu as a Linux Kernel Module. We discuss how SRFD performs

look ahead to avoid potential self-revocation, a long-standing problem in information flow tracking. Our experimental evaluation shows that our approach is efficient, incurring an overhead of a few percentage points, is compatible with existing applications, and provides strong integrity protection.

- **Securing software installation using** SwInst.
  We use SwInst to demonstrate the need for rollback and commit capabilities in an enforcement mechanism, and how these can be utilized to realize highly expressive security policies that cannot be supported otherwise. This system has been successfully evaluated on over 20,000 software packages available on Ubuntu Linux. SwInst covers the initial file labeling during software installation. Spif can then enforce runtime policies based on the initial labeling.

We present related work in Chapter 11. We then conclude the dissertation and discuss future working directions in Chapter 12.

Part I

Enforcement Mechanisms

Chapter 2

Background

Enforcement mechanisms are essential for enforcing security policies, determining the scope, and the range of security policies that can be supported. In this chapter, we discuss enforcement mechanisms for provenance-based policies. To set the context, we first present some of the definitions, terminology, and the threat model in sections 2.1 and 2.2. Then we discuss issues and the challenges for enforcement mechanisms in Section 2.3..

2.1 Terminology

Tracking provenance relies on attaching labels to subjects and objects and enforces policies based on the labels. Specifically, every piece of code and data entering the system needs to be labeled with provenance information. We define provenance as the origin ("where") of a piece of information. In the simplest setting focusing on preserving system integrity, we consider only two provenance labels. Files coming from the OS vendor and any other source that is trusted to be non-malicious are given the label *benign* (Figure 2.1). The remaining files are given the label *untrusted.* The distinction between benign and untrusted is purely based on the trust users have on the sources.

Note that *benign programs* may contain exploitable vulnerabilities, but only *untrusted programs* can be malicious, i.e., may intentionally violate policy and/or attempt to evade enforcement. Exploitation of vulnerabilities can cause benign programs to turn malicious. However, an exploit represents an intentional subversion of security policies, and hence cannot occur without the involvement of malicious entities. Consequently, *benign processes*, which are processes that have never been influenced by untrusted content, cannot be malicious. New files and processes created by benign processes can hence be labeled benign. Processes that execute untrusted code, and those that read untrusted inputs, are labeled as untrusted, as are the files created or written by them. *Trusted programs* are programs that are trusted to handle untrusted inputs at specific interfaces. They are trusted to sanitize untrusted inputs and remain to be benign.

Figure 2.2 (adopted from PPI [Sun et al., 2008b]) shows the classification. Noted that an identifiable subset of benign programs are trusted, while an unidentifiable subset of untrusted programs are malicious.

Term	Explanation
malicious	intentionally violate policy, evade enforcement
untrusted	possibly malicious
benign code	non-malicious but potentially vulnerabilities
benign process	process whose code and input is benign, i.e., non-malicious
trusted process	benign process that can handle untrusted inputs without getting compromised at specific interfaces

Figure 2.1: Key terminology

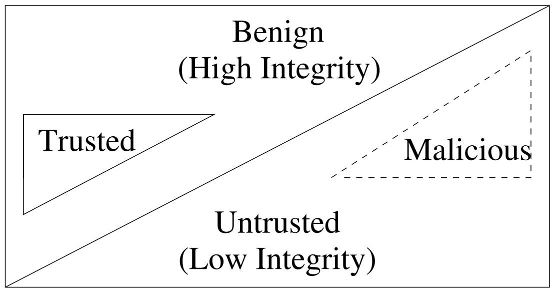


Figure 2.2: Classification of subjects/objects

2.2 Threat Model

We assume that users of the system are benign. Any benign application invoked by a user will therefore be non-malicious. It is possible to designate some users as untrusted, but we believe that in the context of today's malware landscape, users are typically careless or gullible, but not intentionally malicious.

We assume that any file received from unknown or untrusted sources will be labeled as low-integrity. This can be achieved by exclusion: Only files from trusted sources like OS distributors, trustworthy developers, and vendors are labeled as high-integrity. All files from unverifiable origins (including network and external drives) are labeled as untrusted. This labeling convention has been adopted by Windows and OS X. As described later, labeling of incoming files has been seamlessly coupled on the Microsoft Windows OS with Windows Security Zones, which has been adopted by all recent browsers and email clients. For Unix systems, we have developed browser and email client addons to label files. An administrator or a privileged process can upgrade these labels, e.g., after a signature or cryptographic hash verification. A benign process can also downgrade labels.

We first focus on attacks that compromise the system-integrity, i.e., performing unauthorized modifications to the system e.g., malware installing itself for auto-starting or altering the environment to hide or subvert other applications or the OS (e.g., by modifying `bashrc`). Although we can consider protecting confidentiality of user files, this would require confidentiality policies to be explicitly specified. We introduce in Section 8.7 a generalization together with a policy language that can be used to specify confidentiality policies. It should be noted that files containing secrets are useful to gain privileges are already protected from reads by normal users. This policy could be further tightened for untrusted subjects.

2.3 Criteria for enforcement mechanisms

Enforcement mechanisms for provenance tracking place labels on subjects (e.g., processes) and objects (e.g., files), and they enforce policies to restrict how subjects and objects interact. There are a few criteria that enforcement mechanisms need to consider:

2.3.1 Security against evasive malware

Experience with various containment mechanisms such as sandboxie [Sandboxie Holdings, LLC., 2015], Bufferzone [BufferZone Security Ltd., 2015] and Dell Protected Workspace [Dell, 2015], as well as the numerous real-world sandbox escape attacks [Fisher, 2014, Li, 2015, Constantin, 2013] have demonstrated the challenges of building new, effective containment mechanisms for malicious code [Rahul Kashyap, 2013]. Attackers can try every possible way to circumvent the tracking system.

There are two requirements for a provenance tracking mechanisms to be secure:

- **Ability to label resources and propagate labels securely.** Subjects and objects are highly dynamic. A subject can create new subjects and objects. For example, a process may fork or clone itself. Users may create hard links to objects. A provenance tracking enforcement mechanism needs to assign labels to all new subjects and objects based on the corresponding sources in such a way that attacker-controlled and attacker influenced entitles are marked as untrusted.

- **Ability to mediate every interaction between subjects and objects.** OSes support multiple mechanisms for subjects and objects to interact. Subjects and objects can not only interact directly through file reading/writing, but also indirectly via anonymous objects like unnamed pipes, IPC with other subjects using sockets, shared memory, or pipes. Systems like Windows even allow a subject to create a remote thread in another subject. A secure provenance tracking enforcement mechanism needs to mediate every interaction between subjects and objects, or otherwise, attackers could leverage those unrestricted interactions to compromise unconfined processes, take control, and abuse privileges of unconfined processes.

2.3.2 OS Compatibility/portability

Different choices for enforcement mechanisms result in different levels of OS compatibility:

- **Developing a brand new OS:** Enforcement mechanisms that enforce policies at a different OS abstraction would need dramatic modification to OSes. A few research projects (e.g., HiStar [Zeldovich et al., 2006] and Asbestos [Efstathopoulos et al., 2005]) introduce new objects for access control (e.g., memory page or event). Since existing OSes do not support these abstractions, they need to rewrite the entire OS from ground up. We do not see these research OSes getting widely adopted because of the limited application and technical support.

- **Reusing existing OS kernel abstraction:** More often enforcement mechanisms enforce policies using existing OS abstractions at the kernel level, i.e., processes for subjects and inodes for objects. The main advantage of reusing existing OS abstractions for subjects and objects is that the mechanism requires less modification to OSes. Furthermore, OSes already provide some level of tracking for the standard abstractions. Linux LSM [Wright et al., 2002] and TrustedBSD [Watson et al., 2003] already provide hooks throughout the OSes such that callback functions that policies implemented will be invoked to make security decision. This has considerably simplified the task of building a secure enforcement mechanism. However, OS kernels can undergo frequent updates, which often require nontrivial effort towards updating the implementation of enforcement mechanisms.
- **Reusing existing user-level abstraction:** Instead of using abstractions at the kernel level, mechanisms can also enforce policies based on user-level abstractions such as system calls or user permissions. User-level abstractions are relatively uniform across OSes, and remain stable for long periods of time. For this reason, enforcement mechanisms based on user-level abstractions are desirable.

2.3.3 Expressive power and flexibility

Different enforcement mechanisms have different expressive power. A more flexible mechanism could enforce a more powerful policy. The expressive power of an enforcement mechanism comes from how the labels are defined, whether the labels can be changed, and if any recovery mechanisms are available when there are policy violations. We discuss them below.

Provenance labels

There are two main concerns about provenance labels:

- **Number of labels supported**
  Provenance-based tracking systems attach labels to subjects and objects and enforce policies based on the labels. A mechanism supporting

more provenance labels could encode more principals and allow more fine grained tracking and enforcement. In the simplest case, our threat model assumes that there are only two provenance labels: benign and untrusted. Resources therefore have to be labeled as either benign or untrusted. While this simplifies the implementation, it can be too coarse-grained: consider two untrusted origins A and B. Since both origins are not benign, they have to be labeled using the same untrusted label. If A is indeed malicious, it can compromise every resource that shares the untrusted label, including everything belonging to B.

A mechanism supporting more provenance labels could create separate untrusted labels for A and B. Hence, a malicious principal, e.g., A, cannot expand its footprint and compromise other principals.

- **Mapping principals to provenance labels**
  Provenance-based tracking systems regulate how subjects and objects of different provenance labels interact. They need to label and track the effects of the interactions. Provenance labels form a partial order in the system. In the context of integrity, the results would be labeled with the greatest lower bound when subjects and objects interact.

  There are two possible ways to map principals into provenance labels. One is to assign each principal with a provenance label. Any subject or object belongs to exactly one principal. This is less flexible as the provenance label may not reflect exactly the effect that the result actually came from multiple principals. However, it is easier to implement as we do not need to maintain multiple provenance labels for each object and subject.

  A more precise approach is to model labels as representing a set of the principals. This labeling system can then label all possible interactions accurately. The size of the provenance label, however, grows with the number of principals.

Dynamic label changes

Enforcement mechanisms attach labels to subjects and objects, and they enforce policies controlling how subjects and objects can interact. Given a subject with label S and an object with label O with information flowing from the object to the subject, the policy can either permit the operation when $S \leq O$ or deny the operation when $S > O$.

When the enforcement mechanism allows labels to change dynamically, S could change to $S' \leq O$ to permit the information flow. If the subject never writes into a high integrity file after downgrading, the execution would not trigger any policy violation. Allowing subject labels to change could allow more executions to be completed, and hence leads to more usable systems.

On the other hand, if the subject writes to high integrity files after downgrading, the system would flag this as policy violation and deny the writes. If the files were opened before downgrading, applications usually do not handle write failures. This would load to the so-called self-revocation problem. We discuss the problem in more detail in Part II.

Rollback/recovery

While policies focus on deciding how subjects and objects interact, the decisions are often limited by the underlying supporting enforcement mechanisms. When an operation is deemed as violation, the simplest mechanism is to deny the operation during the execution.

Denying an operation during an execution could cause problems. For instance, if application does not expect the operation to fail, it will not handle the failure gracefully. The classical integrity policy low-water-mark has the self-revocation problem, where permission to write to a file could be revoked unexpectedly. This could leave resources in inconsistent states.

The problem that applications cannot handle failures gracefully can be addressed at either the policy level or at the mechanism level:

- **At the policy level**, the policy can deny the operation at a earlier time. The earliest time is to simply deny an application from even running. Other possible time could be at the file opening time. We call these look ahead methods. We leave the discussion to Section 6 (ED/UII) and 5.5 (SRFD). In short, look ahead methods try to deny an operation earlier (promote early failures) so that applications can handle the failures more gracefully. Since it involves predicting future application behavior, policies often need to make conservative decisions and could block some safe executions from completing.

- **At the enforcement mechanism level**, enforcement mechanisms can simply rollback all the changes made by the execution, as if the application has never started. The ability to rollback is much more powerful than at the policy level because it does not require the policy to be conservative.

Chapter 3

Kernel-based provenance enforcement

One approach to enforce provenance policy is to modify OS kernel. Provenance information is encoded using labels for subjects and objects. An important question is how to store these labels. One solution is to store the labels out-of-band, i.e., separated from the objects. SELinux [Loscocco and Smalley, 2001b] and TrustedBSD [Watson et al., 2003] addressed the problem by modifying OSes. They store labels on disks using extended file attributes (ext4) or extension file attributes (HFS). For in-memory entities, LSM uses opaque fields for storing labels, i.e., the labels are stored in-band in these cases.

After defining how to store the labels of subjects and objects, we need to decide on how to mediate interactions between them. We chose the approach of implementing a kernel module using LSM hooks [Wright et al., 2002]. This module defines these hooks in such a way as to maintain provenance, and enforce the specified provenance-based policies. Existing systems such as SELinux [Loscocco and Smalley, 2001b], AppArmor [Ubuntu, 2015], and OS X App Sandbox [Apple Inc., 2014] are all based on the same general approach.

In this chapter, we discuss our approach for building a kernel-based provenance enforcement mechanism. The design of our approach is adopted from [Mital, 2010]. Our main contributions are (1) we implemented the design, (2) we evaluated the performance of the mechanism, and (3) we established the correctness and security guarantee for the mechanism (Section 5.6). We describe our approach below:

3.1 Key abstractions

In our kernel-based enforcement mechanism, there are three key entities: Objects, Subjects and Handles.

Objects Objects consist of all storage and inter-process communication abstractions on an OS: files, pipes, sockets, message queues, semaphores, etc. our approach divides these objects into two categories: file-like and pipe-like. There is a fundamental difference between these classes. File-like objects are persistent, and our approach assigns fixed integrity label to them. Any data read from the file has this label, and writes to the file don't change the label. (The information flow policy ensures that any subject writing to it has a equal or higher label.) For a file-like object, the label of data read from it will be the same as that of data written into it. In contrast, for a pipe-like object, the label of data read from the object representing one end of the pipe is the same as the label of data written to the object representing the other end of the pipe (called a peer object). Examples of pipe-like objects include UNIX pipes and sockets.

Subjects and SubjectGroups Subjects correspond to threads. Since the OS-level mechanisms used in our framework cannot mediate information flows that take place via shared memory, subjects that share memory are grouped into SubjectGroups. SubjectGroups are basically processes. The idea is that all subjects within a SubjectGroup will have the same security label at any time.

Handles Handles is a level of indirection between subjects and objects. They serve to link together objects and subjects that have a unidirectional information flow relationship. There is a many-to-one mapping between handles and subjects, and many-to-one mapping between handles and objects in our approach.

Handles are conceptually similar to file descriptors, but there are some differences as well, e.g., a handle is unidirectional: a handle has either a read or a write capability. (Obtaining both requires two handles.) The label of a read-handle is given by the label of the object that it reads from, while the label of a write-handle is given by the label of the subject holding the handle. When read (or write) operation takes place, our mechanism passes the label of the handle to the corresponding subject (or object).

3.2 Object types

While subjects and handles are largely homogeneous, there are many different types of objects that need to be considered. In order that operations on these objects be handled in a uniform way, we map the actual object operations into several abstract operations as shown in Table 3.1. For the purposes of policy enforcement, some of these operations are either ignored or are treated as a combination of other operations; such operations are shown in italics.

		Operations to																	
		Read/Modify object attribute						Create/ Delete object		Associate handles to objects					Perform reads and writes				
Object Type	Object Subtype	**bind**	**lookup**	**stat**	*unlink*	*rename*	**chmod etc.**	**create**	**delete**	**open**	**close**	*listen*	**accept**	**connect**	**read/recv**	**write/send**	**mmap**	*readfrom*	*sendto*
Files	File	+		+	+	+	+	+	+	+	+				+	+	+		
	Directory	+	+	+	+	+	+	+	+	+	+				+				
Links	Hard link	+	+	+	+	+		+	+										
	Symlink	+	+	+	+			+	+										
Volumes	File sys	+		+				+	+										
Pipes	Pipe			+				+			+				+	+			
	Named pipe	+	+	+	+	+	+	+	+	+	+				+	+			
Sockets	Unix Socket	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+		+	+
	Inet Socket	+		+	+			+		+	+	+	+	+	+	+		+	+
IPCs	Shmem			+				+	+	+	+								
	Other IPCs			+				+	+	+	+				+	+			

Table 3.1: Object types in Linux and the list of abstract operations available on them.

The mapping of concrete operations to the abstract operations may not always be obvious for all object types, so we clarify this below:

- *Files:* We view creation operation as a combination of create and bind operations. The latter requires permission checks corresponding to the directory in which the object is being created. An unlink operation (a rmdir if it is a directory) is treated as a delete on the target object, while a rename is treated as a combination of delete and create.

  Directories are similar to files, and are handled the same as plain files in most cases, but there are some differences as well. For instance, they are not written or mmap'd, although they can be read. A lookup on a directory is treated the same as a read of the directory.

- *Hard links:* These are different from files because they do not have labels associated with them. Although our design could, in principle, associate labels with links, it would be difficult to implement: we rely on extended attributes for storing labels, but there is usually no support for associating extended attributes with links. As a result, permission decisions have to be made on the basis of labels associated with its parent (the directory in which the link resides) and its target (the file pointed by the link). In particular, link creation as well as removal are treated as a bind (to the parent directory) and a write to the target file.

- *Symbolic links:* Since symbolic links are stored as plain files (which contain the name of the target file), labels could be associated with them. Creation and deletion of a symbolic link are both treated as a bind on its parent, whereas a lookup is treated as a read of the link file (but not the target). Symlinks need to be protected and it is possible that the symlink can have the security label different from that of the actual target.

- *File systems:* The only operations on file system are mount and unmount. Note that a mount operation removes the existing interpretation of the mount point, and associates it with a new device. As such, mount is treated as a combination of a remove (of the original directory), a write to the device being mounted (unless it is a read-only mount), followed by a bind. Unmount is similar.

  In the case of mount/unmount operations, additional steps are needed for two reasons. First, the file system being mounted may not be trustworthy, and hence the labels provided by the file system may need to be overridden. Second, some file systems may not be capable of providing labels. To address these problems, we set the device label as the *maximum label* that is possible for any file within the file system represented by the device. In the first case, if the file system associates a label l with a file within it, then we take $glb\_file\_lbl(max\_lbl, l)$ as its label, where glb_file_lbl corresponds to the greatest lower bound operation between the two labels. The glb operation makes sense for integrity: it is the minimum of the integrity level of the file system and the specific label on a file. It also makes sense for confidentiality, since the glb will correspond to the maximum of the confidentialities of the entire file system and the specific file. In the second case, we use *max_lbl* as the default label of all files in the volume. We may need a mount-time option by which the *max_lbl* is set to a value lower than that of object's label.

- *Pipes:* As mentioned earlier, pipes and sockets differ from files in that they represent two distinct object such that data written to one of them can be read from the other and vice-versa. As a result, create and open operations need to be interpreted differently, and appropriate handles associated with the objects.

  Unnamed pipes can be created, but there is no way to delete them. They cannot be opened but can be closed. They cannot be bound to names, and hence do not support operations such as lookup, unlink, rename, chmod, etc. In contrast, a *named pipe* has a name in the directory tree, and hence supports all these operations. In particular, creation of a named pipe implies a bind operation, similar to plain files. Other name-related operations are also handled the same was as regular files (with the exception of how handles are associated with objects).

- *Sockets:* These are very similar to pipes. In particular, *Unix domain sockets* are very similar to named pipes, except for the following differences: (a) bind operation can be separated from creation, (b) handle to object associations are affected by additional operations (accept/-connect), and (c) additional system calls to read/write are available (send/recv). (For datagram oriented sockets, sendto/recvfrom may also be used.)

  Internet-domain sockets differ primarily in terms of the addresses used for binding, and secondarily because LSM provides better hooks for mediating accept and connect system calls in the context of Internet-domain.

- *IPCs:* System V IPCs include those for manipulating message queues, semaphores, and shared memory.

  Shared memory needs to be treated differently because we cannot mediate interactions based on shared memory. Processes sharing memory can be handled as if they have a common mmapped file. Thus, a shared memory creation can be viewed as a combination of a file open (in read-only or read-write mode, based on how the shared memory segment is created), followed immediately by a mmap.

Semaphores and *Message queues* are both handled in the same way, fairly similar to files.

3.3 Label management

By managing provenance labels at the kernel level, these labels can be changed easily and hence supporting downgrading easily. When a subject and an object with different labels interact, apart from denying the operation when it violates the security policy, downgrading could be an extra option to resolve the policy conflict.

However, allowing downgrading to occur at anytime could result in self-revocation, whereby read/write operations on already open files are denied because the label of the subject performing these operations has been downgraded. To prevent self-revocation, our approach limits when downgrading can occur.

Our design uses a novel constraint propagation technique to identify file open operations that introduce a potential for future self-revocations, and denies them. Our design is general, and avoids self-revocation involving files as well as interprocess communication.

The key idea is to deny open operations when a subject already holds open-file-descriptors that can write to high-integrity files. This task is simple enough for stand-alone subjects, but challenges arise when considering processes that interact with each other.

Note that many applications involve processes that communicate via pipes, sockets, shared memory and other IPC mechanisms. If our approach looks at each process in isolation and allows one of them to be downgraded, it is possible that a future read by another process would have to be denied, since it is reading an output of the downgraded process. Since the goal of our approach is to avoid denials of reads/writes, our approach needs better mechanisms to keep track of open file descriptors across collections of processes.

A simple approach to deal with collections of communicating processes is to treat them as a single unit, and downgrade them as a unit. LO-

MAC [Fraser, 2000] uses this approach to avoid self-revocation due to IPC within a UNIX process group. However, LOMAC does not recognize the one-way nature of pipe-based communication, and hence would needlessly downgrade an upstream process when a downstream process opens a low-integrity file. To avoid this, our approach needs a mechanism to keep track of all output files held open by processes that are downstream from each process. Since this information is different for each process, keeping track of it can be messy as well as expensive, especially if the number of processes (or number of open files) grows large.

To overcome these problems, we develop a new approach that is based on propagating constraints about downgradability of processes. In particular, our approach keeps track of the highest integrity of any output file that is held open by a process and any of the processes that it writes to. We call this `min_lbl`. Our approach propagates the `min_lbl` "upstream" through pipes and other communication mechanisms. The result is an approach that relies on maintaining and propagating just this single quantity (`min_lbl`) for each process, instead of having to propagate a large amount of information concerning open file descriptors.

3.3.1 Invariants, Flows, and Constraint Propagation

Our approach maintains a current label (`current_lbl`) field for each object and subject. `current_lbl` is the basis for enforcement. In particular, our approach permits no flow from a source to a destination unless the source's current label is at least equal to that of the destination.

Invariant 1 *Any information flow from an entity A to another entity B must satisfy* `current_lbl`$(A) \geq$ `current_lbl`(B).

Instead of denying the operation when the above invariant does not hold, our approach will attempt to dynamically downgrade the label of the destination. Since our approach restricts downgrading to subjects, B must be a subject, and downgrade occurs when it reads from a handle A. B can protect itself from undesirable downgrades by setting its minimum label (called `min_lbl`). In particular, our approach will not attempt to downgrade `current_lbl` unless the following invariant holds after the downgrade:

Invariant 2 `current_lbl`$(B) \geq$ `min_lbl`(B).

Since our approach does not downgrade the labels of file-like objects, file-like objects will have the same label for `min_lbl` and `current_lbl`. For subjects and pipe-like objects, our approach determines the `min_lbl` by constraint propagation, as described further in Section 3.3.3. Finally, handles do not have an independent value for their current label and minimum label; instead, these are derived from the corresponding values of objects and subjects associated with a handle.

Combining the above two invariants, our approach will permit information flow from A to B in all cases where `current_lbl`$(A) \geq$ `min_lbl`(B). Since self-revocation occurs precisely when such a data transfer is denied, we can say:

Observation 3 *A read (or write) operation that transfers data from an entity A to another entity B will be denied in our approach only if* `current_lbl`$(A) <$ `min_lbl`(B).

3.3.2 Forward information flows

Figure 3.1 illustrates the flow of information between objects and subjects via handles. In this figure, solid lines represent actual flow of information. There are two subjects S_1 and S_2. Flow of information between these two subjects occurs via a socket object O_1 (which is pipe-like), and a file object O_2.

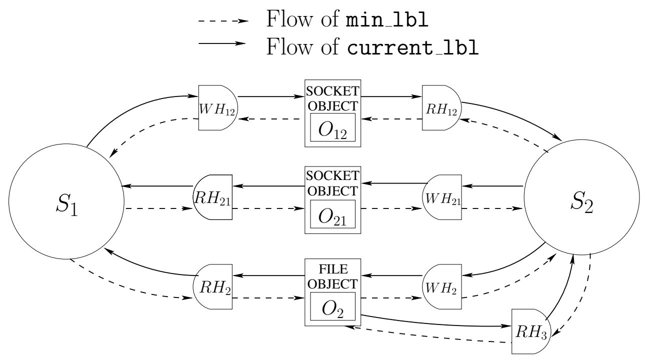


Figure 3.1: Illustration of information flow in our kernel-based framework

Flow of information via file objects is simpler than that of pipe-like objects. In particular, an object created by a subject receives the label of that subject. This flow is handled by propagating the current label of subject S_2 to its write handle WH_2, and then from WH_2 to the object O_2. (If the object is already present, then its `current_lbl` should be less than or equal to that of the subject writing to it, and no propagation would be needed.) If S_1 subsequently reads from the object O_2, the label of O_2 will flow into S_1.

Since a socket is a pipe-like object representing two distinct flows, we split it into two objects: O_{12} that represents information flow from S_1 to S_2, and O_{21} that represents the information flow from S_2 to S_1. S_1 uses a read-handle RH_{21} and a write-handle WH_{12} to read from and write into the socket, while S_2 uses RH_{12} and WH_{21} respectively for the same purpose.

It is important to clarify the role of `open` versus `read` operations. Specifically, when a file is opened for reading, the file's `current_lbl` flows from the file to the handle. But since no data has yet been read by the subject, the propagation of `current_lbl` from the handle to the subject does not take place until the first `read` operation. (A similar comment applies to `write` operations as well.) This distinction between `open` and `read` operations is made for pipe-like objects as well, except that there are many `open`-like system calls, including `pipe`, `connect` and `accept`.

Delaying `current_lbl` propagation serves an important purpose: shells (e.g., bash) often open files for file redirection, and set up pipes for use by its child processes. The shell process does not perform any reads/writes on these objects. By deferring any downgrades until the first `read`, our approach prevents the shell from having to downgrade itself. Such a downgrade of shell's label is disastrous, as it prevents the shell from ever running high-integrity commands.

We note that for memory-mapped files, reads may happen implicitly when memory is read, and hence our approach does not support delayed propagation of labels as described above.

3.3.3 Constraint propagation

As noted earlier, our approach avoids self-revocation by propagating constraints on `min_lbl`. Figure 3.1 shows constraint propagation using dashed lines. Note that constraints propagate in the reverse direction of information flow.

Note that `min_lbl` represents the minimum label that needs to be maintained by a subject A. Any entity B from which information can flow to A needs to maintain a label higher than `min_lbl`(A) or else the flow from B to A may have to be cut-off. Since such cut-offs lead to self-revocation, our approach prevents them by propagating `min_lbl`(A) to any handle from which

A reads; and from this handle to the associated object; and so on. In other words, by propagating `min_lbl` in the inverse direction of information flow, our approach can ensure that every data producer upstream will maintain the integrity level required by A.

Whereas the forward flow of labels is normally delayed until an explicit read or write operation, constraint propagation is instantaneous, i.e., when a channel (representing file or pipe-like communication) for information flow from entity A to another entity B is opened, B's `min_lbl` is propagated immediately to A. Because of Invariant 2, this propagation will fail if A's current label is already less than `min_lbl`(B). In this case, our approach will deny the `open` operation.

It is important to note that `min_lbl` is a quantity that is derived through constraint propagation. It should not be thought of as a variable whose value is increased each time a new communication channel is established. For this reason, `min_lbl` can either *increase* or *decrease* during the lifetime of a subject. Increases happen when a subject opens a new output handle, while decreases happen when a subject closes an output handle.

Due to constraint propagation, the following invariant holds:

Invariant 4 *If there is an information flow path (shown by solid lines in Figure 3.1) from A to B,* `min_lbl`$(A) \geq$ `min_lbl`(B).

Since constraint propagation increases a `min_lbl` value for an entity only if there is a constraint that requires it to be that high, and since files are the only entities that have a hard requirement for their `min_lbl` values, we can make the following observation:

Observation 5 *For an entity A, let $B_1, ..., B_k$ be all the open output files reachable from A while following the information flow paths. Then* `min_lbl`(A) *will be the maximum among* `min_lbl`$(B_1), ...,$ `min_lbl`(B_k).

This observation follows readily from our declarative definition of constraints and their propagation.

3.4 Implementation

We developed our approach on both Ubuntu 13.10 and 14.04. We implemented our approach using the Linux Security Module (LSM) framework.

Although Linux kernel no longer allows loadable modules to use LSM hooks, there are work-arounds available [NTT DATA Corporation, 2010] that we relied on. Structuring the system as a loadable module eases development and debugging, especially in the early stages of prototype development.

The LSM framework provides hooks to mediate system calls and system operations pertaining to inodes, files, tasks, semaphores, shared memory, sockets, and message queues. We present below the LSM hooks that our approach used.

3.4.1 Framework hooks

This section broadly classifies the hooks of our framework based on their purpose. A short description of each hook has also been provided. LSM provides many more hooks than the ones discussed here. However only the pertinent hooks have been used in our implementation, the criteria for selection being the hook's appearance in the sequence of invocation of related hooks, the hook's parameters and the hook's return type.

Much of the work performed in each of the selected hooks falls into the following categories:

- Updating the data structures in response to various operations on subjects and objects; and maintaining the invariants listed in the appendix after each such operation.

- Storing information that is available in one LSM hook so that it can be used in a subsequent hook where it is needed; and more generally, reconstructing information needed by our framework that is not directly available in the LSM hooks.

- Enforcing integrity policies on objects and subjects and making access decisions. It is important to note that a hook with a *void* return type cannot be used for making access decisions.

The classification of LSM hooks used in our framework is as follow:

- **Security hooks for program execution operations**

  - `bprm_committing_creds` : This hook is invoked when a subject executes an object. Our approach uses this hook to downgrade a subject when the object is of a different provenance label.

  - `bprm_check_security` : This hook is invoked when a subject is about to execute an object. Our approach uses this hook to check if information can flow from the object to the subject.

- **Security hooks for filesystem operations**
  - `security_sb_mount` : This hook checks if the runtime binding of a device can occur with a mount point.
  - `security_sb_unmount` : This hook simply checks if the subject can unmount a device.

- **Security hooks for inode operations**
  - `inode_alloc` : This hook is used to allocate an in-memory object security structure to every object represented by an inode and assign a label to it.
  - `inode_free` : This object de-allocates the object security structure and cleans up the memory allocated for its label and handles (if the handles were not already closed).
  - `inode_init_security` : This hook makes the object security structure, associated with the inode, persistent, by writing it on the persistent media (disk), typically in the inode's extended attribute space.
  - `inode_create` : This hook is specifically for regular files and helps the framework perform regular-file specific permission checks (such as `bind`).
  - `inode_link` : This hook is specifically for hard-links and helps the framework perform hard-link specific permission checks (such as `bind`).
  - `inode_unlink` : This hook helps the framework perform permission check (such as `unlink`) on the inode, to remove hard links to it.
  - `inode_symlink` : This hook is specifically for symbolic-links and helps the framework perform symbolic-link creation checks (such as `bind`).
  - `inode_mkdir` : This hook is for directories and helps the framework perform directory creation checks (such as `bind`).

- `inode_rmdir` : The framework uses this to check if a directory can be unlinked from its parent namespace.
- `inode_mknod` : This hook deals with permission checks for creation of special files like pipes and named sockets.
- `inode_rename` : This hook primarily implement the abstract operation
  `rename`.
- `inode_follow_link` : This hook is used for maintaining link traversal information which is used for implementing *virtual downgrades*.
- `inode_permission` : This hook performs the handle creation operation by checking the mode in which the inode is being accessed. The abstract operation `open` is performed in this hook.
- `inode_setattr` : This hook checks permission before setting file attributes.
- `inode_getattr` : This hook checks permission before getting file attributes.
- `inode_delete` : This hook can be used to release any persistent label associated with the inode. Currently this hook is not being used because the clean-up is performed in `inode_free_security`.
- `inode_setxattr` : This hook checks permission before setting the extended attributes.
- `inode_getxattr` : This hook checks permission before getting the extended attributes.
- `inode_removexattr` : This hook checks permission before removing the extended attributes from persistent media.

• **Security hooks for dentry operations**

- `d_instantiate` : This hook is invoked whenever a dentry structure is instantiated for an inode, in the d_cache.

• **Security hooks for file operations**

- `file_permission` : This hook is invoked for every read and write attempted on a file object. Our framework calls the `read` and `write` abstract operations depending on the mode the file is being accessed.

  - `file_ioctl` : This hook checks permission for an ioctl operation on file.
  - `file_mmap` : This hook checks permissions for a mmap operation. Reads and rites to the mmap'ed region are unmediated, this hook helps the framework in setting the desired flag for the read and write handles to the mmap'ed region.
  - `file_fcntl` : This hook checks permission before allowing the file operation specified by the —cmd— parameter from being performed on the file.

- **Security hooks for task operations**
  - `task_create` : This hook is used by the framework to differentiate between fork and clone events.
  - `task_setrlimit` : To perform permission checks on the subject which tries to modify resource limits.
  - `task_kill` : To perform task permission check on one task which is trying to send a signal to another task.
  - `cred_prepare` and `cred_free`: This hook is used by the framework to manage subject and subjectgroup. A new subjectgroup is created when a fork is called. Our approach relies on tracking `cred` structure to relate subjects and subjectgroups because there are no hooks for allocating and destroying subject security structures.

- **Security hooks for Unix domain networking**
  - `unix_stream_connect` : Checks permissions before establishing a Unix domain stream connection.
  - `socket_unix_may_send` : Checks permissions before connecting or sending datagrams from one socket to another.

- **Security hooks for socket operations**
  - `socket_create` : Checks permissions prior to creating a new socket.

  - `socket_bind` : Checks permission before socket protocol layer bind operation is performed and the socket is bound to the specified address.
  - `socket_connect` : Checks permission before socket protocol layer connect operation attempts to connect socket to a remote address
  - `socket_listen` : Checks permission before socket protocol layer listen operation.
  - `socket_accept` : Checks permission before accepting a new connection.
  - `socket_sendmsg` : Checks permission before transmitting a message to another socket.
  - `socket_recvmsg` : Checks permission before receiving a message from another socket.

- **Security hooks for System V IPC Message Queues**
  - `msg_queue_associate` : Checks permission when a message queue is requested through the —msgget— system call.
  - `msg_queue_msgctl` : Checks permission when a message control operation specified by —cmd— is to be performed on the given message queue
  - `msg_queue_msgsnd` : Checks permission before a message is enqueued on the message queue.
  - `msg_queue_msgrcv` : Checks permission before a message is dequeued on the message queue.

- **Security hooks for System V Shared Memory Segments**
  - `shm_associate` : Checks permission when a shared memory region is requested through the —shmget— system call.
  - `shm_shmctl` : Checks permission when a shared memory control operation specified by —cmd— is to be performed on the shared memory region.
  - `shm_shmat` : Checks permissions prior to allowing the —shmat— system call to attach the shared memory segment to the data segment of the calling process.

- **Security hooks for System V Semaphores**
  All operations performed in the following hooks are exactly the same as those performed for the corresponding hooks for System V Shared Memory Segments.
  - `sem_associate`
  - `sem_semctl`
  - `sem_semop`

3.4.2 Code complexity

The overall size of our implementation is shown in Figure 3.2.

	C	Header	Python	Total
Kernel Code	3844	865	-	4709
Userland code	643	142	57	842
Total	4487	1007	57	5561

Figure 3.2: Implementation code size for our kernel-based enforcement approach

Our approach has a userland component that can communicate with the kernel module using netlink to manage policies. The userland code will be notified when there is a policy violation. The userland can then enforce additional policy on how to handle the violation, which is a topic of Part II.

3.5 Alternative: one-way isolation

Isolation is another useful policy that the kernel can enforce. There are two types of isolation: one-way isolation and two-way isolation. Virtual machines and browsers SOP generally enforce two-way isolation. They prevent resources from one provenance to interact with another. On desktop environment, this can be implemented using virtual machines or LXC [Canonical Ltd., 2012] so that applications running within a container cannot access resources on the host.

One-way isolation [Sun et al., 2005, Liang et al., 2009] permits untrusted software to read shared resources, but its outputs are held in isolation. It is usually implemented using copy-on-write file systems. To separate the process and IPC namespace, LXC is also used, and mounts the host file system as read only.

Although App model and SOP also use isolation, their usage is for isolating resources of each provenance. When applying in the context of provenance-based enforcement, one-way isolation is far more powerful when applied as a rollback mechanism to recover from policy violation and enforce state-based policies.

Supporting rollback

One of the main disadvantages of enforcing provenance-based policies is that policy violations can occur at a point when it is hard to recover. For example, the application may not expect an operation to result in a security failure (e.g., read operations on open file descriptors) and hence may terminate the execution abruptly upon policy violation. This could leave the system in an inconsistent state. Provenance-based policies therefore focus on detecting potential policy violations earlier, and thus promote early failures.

By running an execution within a one-way isolation environment, the system can buffer the changes that an execution would have. If there is a policy violation, the system can simply discard the changes. Otherwise, the changes can be committed and made visible to the rest of the system. The policy need not to be conservative to promote any failure.

One-way isolation alone is not enough to support a more general provenance-based policies (i.e., confidentiality). It imposes no restriction on reading. To enforce confidentiality policies, the system needs to monitor read accesses. If processes attempt to read any confidential file, the enforcement mechanism can terminate the execution immediately.

Chapter 4

Userlevel-based provenance enforcement

Provenance tracking and enforcement techniques described above, as well as those developed by previous research require significant changes to the OS kernel [Li et al., 2007, Sun et al., 2008b, Mao et al., 2011], or the development of brand new OSes [Zeldovich et al., 2006, Efstathopoulos et al., 2005]. Developing such a system-wide tracking mechanism can be error-prone and involve substantial engineering challenges. This problem is particularly serious in the context of closed-source OSes such as Windows. We therefore develop an approach for provenance tracking and secure policy enforcement using security mechanism that are universal to today's desktop OSes, namely, multi-user protection and discretionary access control (DAC). A key strength of our approach is its simplicity and portability, enabling its implementation on Linux, FreeBSD as well as all modern versions of Microsoft Windows (Windows XP through Windows 10).

We focus our discussion on building our approach with only two provenance labels (principals) called benign and untrusted. Our approach extends the idea of mobile OSes to desktop OSes, and is designed to protect legacy desktop applications. We have implemented our approach on multiple OSes, including Linux, BSD, and Windows. It is compatible with existing applications such as Firefox, Internet Explorer, Chrome, Microsoft Office, Adobe Reader, and Photoshop.

4.1 Approach overview

Secure enforcement and tracking without OS changes.

To reliably track provenance, our approach uses an existing security mechanism, namely, multi-user protection and discretionary access control (DAC). Unlike Android, which uses a different userid for each app, our design creates one new userid for each existing user. While Android's goal is to isolate different apps, we use DAC to protect benign processes/files from untrusted code/data. (We discuss the alternative of using Windows integrity labels in Section 8.6.7.)

Our approach introduces a set of untrusted userids. It encodes provenance labels into file ownership and permission. In particular, untrusted files are those that are owned by the untrusted userids, or are writable by these users. Untrusted processes are all run with an untrusted userid. This encoding enables us to leverage existing OS mechanisms for tracking and propagating provenance labels. In particular, note that files as well as child processes inherit their ownership from that of the process that created them. As a result, any file or process created by an untrusted process will have an untrusted label.

Benign processes and files are characterized by their ownership by a userid other than an untrusted userid. In addition, benign files will have write permissions that make them unwritable by untrusted userids by default. This provides the basic protection to benign processes against untrusted processes. Files created by benign processes will have benign labels, once again ensuring correct propagation of labels. Benign processes aware of our approach can request our system to mark a benign file as untrusted.

In addition to tracking provenance labels, our userid-based encoding also provides the foundation for sound policy enforcement without OS kernel changes. Specifically, existing OS mechanisms can correctly enforce basic policies on untrusted processes: by virtue of our provenance label encoding, benign files have permission settings that make them unwritable by untrusted userids. As benign processes can be vulnerable, our approach supports enforcing policies on benign processes as well, e.g., to prevent them from reading untrusted files. This is a considerably simpler task than policy enforcement on untrusted code. In particular, challenges in secure policy enforcement arise mainly due to evasion attacks. Since benign processes cannot be malicious, they won't attempt evasion. Indeed, a simple yet secure implementation can be developed within the address space of a benign process, e.g., by replacing libc/ntdll.dll, which makes all system calls on behalf of a process, with a version that enforces the desired policies.

Application transparency.

Our approach applies a novel *dual-sandbox architecture* to achieve secure enforcement. The first of these sandboxes performs eager policy enforcement. To minimize breaking legitimate functionality, it blocks only those operations that can cause irreparable damage, e.g., overwriting an existing benign file. This sandbox, called untrusted sandbox (U), needs to be secure against any attempts to circumvent it.

Operations with unclear security impact, such as the creation of new files, are left alone by the second sandbox, called benign sandbox (B). While these actions could very well be malicious, there isn't enough information to make that conclusion with confidence by U. Hence, we rely on B to observe subsequent effects of this action to determine if it has to be stopped. Policies will have much more information to decide. For instance, B would prevents a benign process from using files that could compromise the benign process.

Our dual-sandbox architecture preserves functionality of both benign and untrusted applications by implementing many important transparency features, so that security benefits of our approach can be achieved without requiring changes to applications, or the way in which users use them.

Since our approach labels processes using userid, our approach treats applications as blackboxes and requires no application modification. Applications already support running with different userids natively. Our approach can therefore support feature-rich unmodified applications such as Photoshop, Microsoft Office, Adobe Reader, Windows Media Player, Internet Explorer, and Firefox.

Implementation on contemporary OSes.

We have implemented our approach on Linux, BSD, and Windows, supporting XP, 7, 8.1, and 10. Implementing such a system-wide information flow tracking system on closed-source OSes is challenging. We present different design choices we made during the development of our approach. We share our experiences and lessons on implementing it on different OSes such that researchers can be aware of the techniques we applied, and start developing defenses on popular OSes.

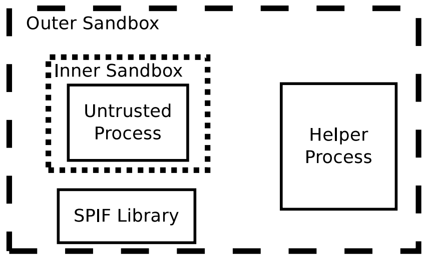


Figure 4.1: Untrusted sandbox

4.2 Containing Untrusted Processes

Our approach leverages existing userid mechanisms in OSes to track provenance information. It confines both benign and untrusted processes. In this section, we focus the discussion on untrusted sandbox.

Our untrusted sandbox, illustrated in Figure 4.1, consists of a simple inner sandbox U_I based on OS-provided access control mechanisms, an outer sandbox that is realized using a library U_L, and a user-level helper process U_H.

The inner sandbox U_I enforces a basic isolation policy that limits untrusted processes so that they can only perform operations that do not affect benign processes, e.g., write to untrusted files. This strict mechanism, by itself, can cause many untrusted applications to fail. For example, an untrusted document writer cannot create temporary files or save files on user's desktop. The outer sandbox is designed to relax the restrictions imposed by U_I. The transparency library U_L component of the outer sandbox masks these failures so that applications can continue to operate as if they were executing directly on the underlying OS. In particular, U_L remaps some of the failed requests (primarily, system calls) so that they would be permitted by U_I. As U_L runs in the untrusted context, U_L may not resolve all failures due to the DAC permission. In those cases, U_L forwards the request to U_H, which runs with the userid of a normal user, to carry out the request. The helper U_H uses a basic default policy that is more permissive than the inner sandbox, but it can also enforce a different policy.

In addition to modifying or relaying requests from untrusted processes, the transparency library U_L also supports the following two remapping mechanisms for file accesses:

- *Shadowing:* Instead of modifying the actual copy of the file, U_L can transparently shadow the access to a private copy of the file.

- *Redirection:* Instead of creating a new file at the intended location, U_L can transparently redirect the file creation to a different location. Future accesses to the file will be redirected as well.

Whether a particular file access is shadowed or redirected can be specified in the policy, a topic further discussed in Section 6.

By splitting the untrusted sandbox into inner and outer sandboxes, our approach can rely on existing OS mechanism to enforce a non-circumventable policy against malicious code. The outer sandbox is circumventable, but bypassing it does not let untrusted code gain any privilege. The inner sandbox only needs to deny untrusted processes from accessing resources owned by benign users. This allows our approach to be deployed on most multi-user OSes with users as basic trust units. OSes such as Windows support advanced user permission models, e.g., ACLs, can grant untrusted processes preciously the safe accesses that they can have. For these OSes, the outer sandbox needs not be separated from the inner sandbox.

Inner Sandbox U_I

Contemporary desktop OSes provide access control mechanisms for protecting system resources such as files, registry entires and IPCs. Moreover, processes belonging to different users are already isolated from each other. We repurpose this mechanism to realize the inner sandbox. Such repurposing would, in general, require some changes to file permissions, but our design was conceived to minimize such changes: our implementation on Ubuntu Linux required changing permissions on less than 60 files (Section 4.5). Windows ACL supports permission inheritance and hence only a handful of top level directories and registry entries needed modification. Moreover, this DAC permission overloading preserves all of the functionality relating to the ability of users to share access to files.

The basic idea is to run untrusted processes with newly-created users that have very little, if any, direct access to modify the file system. For each non-root user[1] R in the original system, we add a corresponding untrusted user

[1]We don't support untrusted code execution with administrative privileges. In Chapter 3, we described a kernel-based system which supports running untrusted code as root.

```
create a new group $G_B$
for each user $U$ do
    add a userid $U$ to group $G_B$
for each real user $R$ do
    create a userid $R_U$ for the user $R$
for each group $G$ do
    create a new group that $G_U$ with the members of $G$
    for each user account $R$ in $G$ do
        add $R_U$ to $G_U$
```

Figure 4.2: Algorithm for setting up users in dual-sandboxing approach

R_U. Similarly, for each existing group G, we create an untrusted group G_U that consists of all userids in G and their corresponding untrusted userids. To further limit accesses of R_U, we introduce a new group G_B of existing ("benign") userids on the system before untrusted userids are added. File permissions are modified so that world-writable files and directories become group-writable by G_B [2]. Similarly, world-executable *setuid* programs are made group executable by G_B. The algorithm is presented in Figure 4.2.

With the above permission settings, no R_U will have the permission to create or modify any file in the system. To support redirection and shadowing, our approach creates a redirect and a shadow directory for each R_U so that R_U can create or modify objects inside. Shadowing consists of reading the original copies and creating shadow copies. By granting R_U permissions to create files, at least half of the logic for shadowing can be offloaded to R_U. This greatly simplifies the complexity of the outer sandbox.

Our mechanism ensures that benign processes will not consume untrusted files by ensuring that they do not access objects within the redirect or shadow directory. Since untrusted files are either redirected or shadowed, benign processes are not even aware of the untrusted files by default. Untrusted processes cannot modify benign files either, since the benign sandbox ensures appropriate permission settings on benign files during creation.

Untrusted processes can compromise benign processes through inter-process communication. Some communication mechanisms, such as pipes between parent and child processes, need to be closed when a child process of a benign process becomes untrusted. This can happen in our system only through

[2]If group permissions are already used, then we use ACLs instead.

the `execve` system call or `CreateProcess` Windows API. Other communication mechanisms such as signals and IPC are restricted by the OS based on userids, and hence the inner sandbox will prevent them already. For intra-host socket communication, the benign sandbox is responsible for identifying the userid of the peer process and blocking the communication. While our mechanism can also rely on the untrusted sandbox to prevent untrusted processes from connecting to benign sockets, some OSes such as BSD do not honor permissions on sockets. Hence, our approach places checks on the benign sandbox at the time of connection establishment. To block communication with external hosts, appropriate firewall rules can be used, e.g., using the `uid-owner` and `gid-owner` options provided by `iptables`.

Using userid as an isolation mechanism has been demonstrated in systems like app model on Android for isolating applications. One of our contributions is to develop a more general design that not only supports strict isolation between applications, but also permits controlled interactions. (Although Android can support interactions between applications, such interactions can compromise security, providing a mechanism for a malicious application to compromise another benign application. In contrast, our approach ensures that malicious applications cannot compromise benign processes.) Our second contribution is that our approach requires no modifications to (untrusted or benign) applications, whereas Android requires applications to be rewritten so that they do not violate the strict isolation policy.

Transparency Library U_L

Operations such as requesting the helper, performing shadowing and redirection cannot be encoded as permission. Our approach uses U_L to modify the behaviors of system library to support these operations. Note that U_L operates with the same privileges as the untrusted process, so no special security mechanisms are needed to protect it. Specifically, U_L handles the following transparency issues:

Userid and group transparency Applications may fail simply because they are being run with different user and group ids. For this reason, U_L wraps `getuid`-related system calls to return R for processes owned by R_U. It also wraps `getgid`-related system calls to return G for processes group-owned by G_u. On UNIX, this mapping is applied to all types of userids, including effective, real and saved userids. As a result, an untrusted process is not even aware that it is being executed with a different userid from that of the user invoking it.

This modification is important for applications that query their own user or groupid, and use them to determine certain accesses, e.g., if they can create a file in a directory owned by R. If not, the application may refuse to proceed further, thus becoming unusable. Some common applications such as `OpenOffice`, `gedit`, `eclipse` and `gimp` make use of their userid information. U_L ensures that such applications remain usable. This modification is also crucial for enhancing usability on Windows— shortcuts such as `Desktop` and `My Documents` in file selection dialog boxes are pointed to R rather than R_U, despite the fact that R_U is running the processes.

File access transparency Both shadowing, redirection, and merging of directory contents are implemented in U_L by intercepting calls to file creation/open, `getdents`, or `NtQueryDirectoryFile`. Untrusted processes can rely on existing DAC permission to read and shadow files on their own. If untrusted processes do not have permissions to read the original files, they can request U_H to get access to the file and shadow the file themselves. Note that our approach implemented its own COW semantics as existing COW semantics would not allow R_U to create files inside R's shadowed directories.

Helper Process U_H

In the absence of our approach, programs will be executed with the userid R of the user running it. Thus, the maximum access they expect is that of R, and hence U_H can be run with R's privileges.

Observe that the inner sandbox imposes restrictions (on R_U relative to R) for only three categories of operations[3]: file/registry/IPC operations, signaling operations (e.g., `kill` or `CreateSemaphore`), and tracing operations (e.g., `ptrace` or `CreateRemoteThread`). We have not found useful cases where R_U needs to signal or trace a process owned by R. Registry entries and IPC objects support permission settings, and hence our approach treats them the same as files. Consequently, we focus the discussion on file system operations that U_H enforces by default:

[3]Recall that R cannot be `root`, and hence many system calls (e.g., changing userid, mounting file systems, binding to low-numbered sockets, and performing most system administrative operations) are already inaccessible to R-processes. This is why it is sufficient to consider these three categories.

- *Read-permission:* By default, R_U is permitted to read every object (file, registry, pipe, etc.) readable by R. This policy can be made more restrictive to achieve some confidentiality objectives. We leave the discussion in Chapter 8.7.

- *Write-permission:* By default, R_U subjects are *not* permitted to write objects that are owned by R. However, instead of denying, untrusted sandbox can shadow the accesses transparently by copying the original file F to R_U's shadow directory. Henceforth, all accesses by R_U-subjects to access F are transparently redirected to this shadow file.

  Shadowing enables more applications to successfully execute by avoiding permission denials. But this may not always be desirable, as it can create confusion to users as there can be multiple copies of the same file. It is sometime desirable to deny the operation. We describe in Chapter 6 how to decide between denial and shadowing.

- *Object creation:* New object creation is permitted if R has permission to create the same object. R_U creates these new objects in redirected directory and benign processes will not be permitted to read them. If R creates an object whose name collides with an untrusted object, either file exists error will be returned or the untrusted object will be shadowed, depending on the type of the object. The policy is detailed in section 6.

- *Listing directories:* As R_U's files are either redirected or shadowed, benign and untrusted files are separated. This can lead to usability problem as file system namespace is fragmented. To preserve a unified file namespace, Our approach merges the contents from the directories transparently as in unioning file system.

- *Operations to manipulate permissions, links, etc.:* These operations are handled similar to file modification operations: if the target file(s) involved is untrusted, then untrusted processes can perform the changes as permitted by the inner sandbox.

- *Operations on R's subjects:* R_U-subjects are not allowed to interact with R-subjects. These include creating remote threads in or sending messages to R's processes, or communicating with R's processes using shared memory.

- *Other operations:* R_U-subjects are given the same rights as those of R for the following operations: executing files, querying registry, renaming untrusted files inside redirected/shadow directory, and so on. Operations that modify benign file attributes are denied by U_I by default.

Note that it is possible that a file may be at a shadowed directory, main file system, or both. Users may have a hard time locating such files, as untrusted copies are visible only to untrusted processes. A policy can specify what files to be shadowed. This is further discussed in Section 6.

While we do not emphasize confidentiality support in this chapter, our approach provides the mechanism for sound enforcement of confidentiality restrictions by tightening the policy on user-readable files. We discuss more in Section 8.7

Windows implementation

In our Windows implementation, our approach grants all of the above rights, except for shadowing and redirection, to R_U-subjects by configuring permissions on objects accordingly. There is no need for U_H. Unlike UNIX, object permissions on Windows are specified using ACLs, which can encode different accesses of arbitrary number of principals. Moreover, there are separate permissions for object creation versus writing, and permissions can be inherited, e.g., from a directory to files in the directory. These features give our approach the flexibility to implement the above policies. Shadowing and redirection are implemented using U_L. Both reading the original files and creation of untrusted files in the shadowed/redirected directories can be completed by untrusted processes themselves.

4.3 Protecting Benign Processes

Our benign sandbox completes the second half of our sandbox architecture. Whereas the untrusted sandbox prevents untrusted processes from directly damaging benign files and processes, the benign sandbox is responsible for protecting benign applications from indirect attacks that take place through input files or inter-process communication.

Existing mechanisms focus on restricting only untrusted processes: policy-based confinement mechanisms focus on sandboxing untrusted processes;

Windows Integrity Mechanism (WIM) enforces no-write-up policy to protect benign processes from being attacked by untrusted processes. However, they do not enforce any policy on benign processes. A benign process can read untrusted files and then get compromised. This is well illustrated by the *Task Scheduler XML Privilege Escalation attack* [jduck, 2014] in Stuxnet, where a user-writable task-file is maliciously modified to allow the execution of arbitrary-commands with system privileges. Hence, it is important to protect benign processes from consuming untrusted objects accidentally.

While policy enforcement against untrusted processes has to be very secure, policies on benign subjects can be enforced in a more cooperative setting. Benign subjects do not have malicious intentions, and hence they can be trusted *not* to actively circumvent enforcement mechanisms[4].

In this cooperative setting, it is easy to provide protection— our approach uses a benign sandboxing library B_L that operates by intercepting calls to system calls used for making security-sensitive operations and changing their behavior so as to prevent attempts by a benign process to open untrusted objects. In contrast, a non-bypassable approach will have to be implemented in the kernel, and moreover, will need to cope with the fact that the system call API in Windows is not well-documented.

A simple way to protect benign applications is to prevent them from ever coming into contact with anything untrusted. However, total separation would preclude common usage scenarios such as the use of benign applications (or libraries) in untrusted code or the use of untrusted applications to examine or analyze benign data. In order to support these usage scenarios, our approach provides three basic interaction mechanisms in three categories as follows. The policy can decide which mode to use, and what other modes

- *Logical isolation:* By default, benign applications are isolated from untrusted components by the benign sandbox, which denies any attempt to open an untrusted file for reading, or engaging in any form of interprocess communication with an untrusted process.

- *Unrestricted interaction:* The other extreme is to permit benign applications to interact freely with untrusted components. This interaction

[4]Although benign applications may contain vulnerabilities, exploiting a vulnerability requires providing a malicious input. Recall our assumption that inputs will be conservatively tagged, i.e., any input that isn't from an explicitly trusted source will be marked as untrusted. Since a benign process won't be permitted to read untrusted input, it follows that it won't ever be compromised and hence won't actively subvert policy enforcement.

is rendered secure by running benign applications within the untrusted sandbox.

- *Controlled interaction:* Between the two extremes, benign applications may be permitted to interact with untrusted processes according to the policy module, while remaining as benign processes. Since malware can exploit vulnerabilities of benign software through these interactions, they should be limited to *trusted* programs that can protect themselves in such interactions.

The first and third interaction modes are supported by a benign sandboxing library B_L. As described in Section 4.3, the default policy relies on B_L to enforce policies to protect benign processes from accidental exposure to untrusted components. The second interaction mode makes use of the untrusted sandbox described earlier, as well as a benign sandboxing component (Section 4.4.1) for secure context switch from benign to untrusted execution mode. Our approach supports enforcing more sophisticated policies.

Benign Sandboxing Library

Since benign processes are non-malicious, they can be sandboxed by enforcing policies using library. By default, our approach enforces the isolation mode, B_L enforces the following basic policies. More advanced policies can be specified and enforced by B_L

- *Listing directories:* Attempts to list directories that have been redirected will be merged to present users an unified view by default. This is the only operation that untrusted files are involved. It is a trade-off between security and usability— our approach assumes that benign processes cannot be compromised simply because of the presence of an untrusted file. This allows users to know the existence of untrusted files and infers user intentions to transition to untrusted domain.

- *Querying file attributes:* Operations such as `access`, `stat` and `NtQueryAttributesFile` that refer to untrusted files are denied. The default error is file not exist, as untrusted files are shadowed/redirected. However, policies can specify a customized error if such error can communicate security failures to the users more meaningfully.

- *Executing files and reading files/registry entries:* These are handled in the same way as file attribute query operations. Since untrusted files and registry entries can be shadowed or redirected, this checking involves examining the paths to the system call.

- *Opening non-file objects for reading:* Our appropriate does not permit benign processes to read untrusted objects by default. These opens will be denied as in the querying file attributes operations. To avoid race conditions, the object needs to be opened and a stat is performed on the object descriptor/handle. Note that post-checking is necessary for non-file objects because our approach does not separates namespace for untrusted non-file objects, as not all OSes support object namespace.

- *Changing file permissions:* These operations are intercepted to ensure that benign files are not made writable to untrusted users accidentally by default. These restrictions prevent unintended changes to the provenance labels of files. However, there may be instances where a benign process output needs to be marked as untrusted. Our approach provides both an utility and specific library calls for this purpose. Our approach uses DAC permission to ensure that only benign processes can invoke this utility.

- *Interprocess communication channel establishment:* This includes operations such as `connect` and `accept`. The OS is queried for the userid of the peer process. If it is untrusted, the communication will be closed by default, and our approach returns a failure code. For OSes that do not support querying userid based on sockets, our approach can incorporate in-band authentication mechanisms to challenge peer's identity right after the channel establishment. The challenge can be as simple as asking the peer process to return the content of a user-readable file which untrusted users cannot read.

- *Loading kernel modules:* Similar to opening files for reading, untrusted modules or drivers are redirected and hence are not visible to benign processes for loading into the kernel.

In addition to isolation, B_L can also support controlled interaction between benign and untrusted processes. This option should be exercised only with *trustworthy* programs that are designed to protect themselves from malicious inputs. Moreover, *trust should be as narrowly confined as possible,* so B_L can limit these interactions to specific interfaces and inputs on which a benign application is trusted to perform sufficient input validation. To highlight this aspect, we refer to this mode of interaction as *trust-confined execution.*

B_L provides two ways by which trust-confined execution can deviate from the above default isolation policy. In the first way, an externally specified policy identifies the set of files (or communication end points such as port numbers) from which untrusted inputs can be safely consumed. We discuss more in an application of our approach, namely secure software installation, in Section 10. The policies can also specify if certain outputs should be marked as untrusted. In the second way, a trusted process uses an API provided by B_L to explicitly bypass the default isolation policy, e.g., `trust_open` to open an input file even though it is untrusted. While this option requires changes to the trusted program, it has the advantage of allowing its programmer to determine whether sufficient input validation has been performed to warrant trusting a certain input.

4.4 Switching Between Benign and Untrusted Contexts

4.4.1 Context-Switching at Process Execution Time

Users may wish to use benign applications to process untrusted files. Normally, benign applications will execute within the benign sandbox, and hence won't be able to read untrusted files. To avoid this, they need to preemptively downgrade themselves and run within the untrusted sandbox. Our approach provides a mechanism for downgrading subjects at the exec time. The (policy) decision as to whether to downgrade this way is discussed in Chapter 6.

Switching security contexts (from untrusted to benign or vice-versa) is an error-prone task. For a benign process to run an untrusted program, it needs to change its userid from R to R_U.

Transition on UNIX One of the advantages of our design is that it leverages a well-studied solution to this problem, specifically, secure execution of setuid executables in UNIX.

A switch from untrusted to benign domain can happen through any setuid application that is executable by untrusted users. This transitioning can be useful when untrusted processes need to perform actions that are only available to benign processes. Well-written setuid programs protect themselves from malicious users. Moreover, OSes incorporate several features for protecting setuid executables from subversion attacks during loading and initialization.

Transitions in the opposite direction (i.e., from benign to untrusted) are far more common and require more care because processes in untrusted context cannot be expected to safeguard system security. We therefore introduce a gateway application called `uudo` to perform the switch safely. Since the switch would require changing to an untrusted userid, `uudo` needs to be a setuid-to-root executable. It provides an interface similar to the familiar `sudo`[5] program on UNIX systems — it interprets its first argument as the name of a command to run and the rest of the arguments as parameters to this command. By default, `uudo` closes all benign files that are opened in write mode as well as IPC channels. These measures are necessary since all policy enforcement takes place at the time of `open`, which, in this case, happened in the benign context. Next, `uudo` changes its group to G_U and userid to R_U and executes the specified command. (Here, R represents the real userid of the `uudo` process.)

Transition on Windows On UNIX, the transition can be performed using `setuid`, but Windows only supports an impersonation mechanism that temporarily changes security identifiers (SIDs) of processes. This is insecure for confining untrusted processes as they can re-acquire privileges. The secure alternative is to change the SID using a system library function `CreateProcessAsUser` to spawn new processes with a specific SID. Our approach uses a Windows utility `RunAs` to perform this transition. `RunAs` behaves like a setuid-wrapper that runs programs as a different user. It also maps the desktop of R_U to the current desktop of R so that the transition to user R_U is seamless. By passing `RunAs` with the appropriate parameters, it serves the purpose of `uudo`.

[5]The name `uudo` parallels `sudo`, and stands for "untrusted user do," i.e., execute a command as an untrusted user. The most typical usage scenario is to start an untrusted shell by executing `uudo bash` or `uudo cmd`.

4.4.2 Dynamic (Any-time) Context-Switching

In this section, we describe how we can leverage existing constructs in OSes to build a dynamic downgrading system without any kernel modification. The mechanism described in Chapter 3 maintains labels in the kernel, which provides a lot of information about processes and flexibility for policy enforcement and label management. It can therefore support dynamic downgrading easily. Enforcing similar policy in the user-space is challenging, yet it would be more robust and easily deployable on various Unix based OSes.

Downgrading mechanism

Supporting dynamic downgrading requires the ability to change provenance labels of processes dynamically during their executions. For our user-space approach, this will requires changing userids of processes dynamically. Unix supports `setuid` system calls that allow processes to change userids. Typically, setuid is used by root processes to switch to another users. For example, the `login` program runs as root when the system starts. Upon receiving and authenticating user login credentials, `login` will call `setuid` to change ownership to the user. All processes spawned by `login` will then automatically inherit the credentials of the user.

While privileged processes can arbitrarily switch their userids using `setuid`, normal user processes cannot do so. UNIX setuid mechanism was developed to address this very problem, but it can only be used at the time of execve[6]. Accomplishing userid changes at other times seems impossible from the design of UNIX mechanisms, but we show below how we can achieve this, by a clever exploitation of the specifics of userid mechanism.

There are three different userids associated with each process. They are ruid (real userid), euid (effective userid), and suid (saved userid). By convention, ruid represents the identity of the logged in user ("real user") whose actions spawned this process. Security decisions, in contrast, are made on the basis of euid. suid is designed for processes to store userids that they can use later. However, most processes have suid the same as euid or ruid.

For most processes, these three userids share the same value. Processes that have different ruid and euid are typically those running setuid binaries or root-owned processes. When a process executes a setuid binary, the ruid of

[6]This is the reason for discussing execve-time context switching separately in the previous section.

the process will remain as the ruid of the process. This allows setuid programs to check who executed the programs. However, euid of the process will be changed to the owner of the setuid binary. Since all permission checking is based on euid, this allows the process to be granted the privileges of the setuid binary owner[7].

Apart from the `setuid` system call that allows a root process to change its userids to arbitrary values, there are other setuid system call variants for manipulating userids. Specifically, `setresuid` allows a process to change its ruid, euid and suid to any of its ruid, euid and suid. For most user processes that have the same ruid, euid and suid, calling `setresuid` has no effect.

By exploiting the semantics of `setresuid`, we can extend our approach to achieve userid-based dynamic downgrading. `setresuid` provides a mechanism for processes to change ownership. In our system, benign user processes will be running with two userids: real user and "untrusted" user. As long as the process is benign, the process will have privileges of the real user. To downgrade a process, our system simply needs to change all the userids of the process to the "untrusted" user (or drop the benign userid). As a result, the process can no longer have the privileges of the benign user.

Propagation and maintenance of userids

To support dynamic downgrading, processes need to be spawned with two userids. However, as discussed previously, privileged processes like `login` call `setuid` to set ruid, euid and suid all to the actual user's userid. Child processes simply inherit userids from parent processes and hence have only one userid. One way to support multiple userids for all user processes is to modify the behavior of the privileged processes such that they maintain two userids. Specifically, we can convert calls to `setuid` into `setresuid` to maintain two userids. Then all the child processes will automatically have two userids and can be downgraded. However, most processes, e.g., window managers or file managers, do not downgrade and can simply run as usual with a single userid.

Alternatively, we can grant processes two userids only if they might need

[7]A setuid program running with the program owner privilege may want to restrict it accesses just to what the real user has. On Linux, `access` system call checks the privileges against ruid. Hence, a root process may not be allowed to `access` a user file. Typical programs do not expect to run in setuid context simply uses `access` for checking privileges of the current users.

to be downgraded. Instead of intercepting the transition from root to user at the `setuid` call, we can grant processes two userids at the time of the `execve` system call. This can be achieved by executing a setuid-to-root binary whenever a process needs to have the downgradability privilege. Instead of executing the desired program image, our system executes the setuid-to-root program. This setuid-to-root binary is nothing but a program to execute the command specified in the parameter similar to `uudo`. Upon executing this binary, the process privilege will be escalated to root and having euid = 0. The program can then invoke `setresuid` to set ruid, euid and suid to contain the two userids before executing the actual program images. Since running this setuid-to-root binary allows any process to become downgradable process, this setuid binary needs to be protected so that only benign processes can execute.

Granting two userids by executing a setuid-to-root binary has some limitations. Specifically, the loader automatically discards some of the environment variables when executing setuid-to-root binaries because environment variables such as LD_LIBRARY_PATH and LD_PRELOAD can compromise the execution of the setuid process and lead to privilege escalation attacks. Relying on exec-time to grant two userids could therefore affect the functionality of applications.

Another way to grant processes two different userids is by inheriting the two userids from their parent processes during exec. It is therefore important to understand the semantics of `exec` when processes have multiple userids. When a process executes an image, its suid will be overwritten by the process's euid. Only processes' ruid and euid are preserved when `exec` is called. This is because suid is considered for program's internal use and hence is not preserved across `exec`. As a result, our system cannot use suid for propagating the two userids.

Since a process with two userids represents a benign, downgradeable process in our design, the euid of this process should be made to correspond to a benign user so that it is granted the privileges of a benign user. That leaves only the ruid field available to store an untrusted userid. Unfortunately, using ruid in this manner violates the intended semantics of ruid, which leads to compatibility problems. For instance, applications frequently use `access` system call to verify user privileges. If we store an untrusted userid in the place of ruid, such applications will receive incorrect information about the actions permissible for a process. It may seem possible to address this incompatibility by transparently replacing a call to `access` with an `eaccess` that uses euid instead of ruid. Unfortunately, this does not work either, because `eaccess` is not compatible with systems that use ACLs in addition to conventional UNIX file permissions.

Considering the complications mentioned above, we devised an approach that stores the untrusted userid into suid normally, but moves it into ruid during exec. When executing a setuid-to-root binary, the process will have ruid as untrusted and euid representing the real user. After executing the desired program image, we can then have suid set to untrusted, and both of ruid and euid set to that of the real user. This approach preserves the semantics of the userids: ruid and euid remain unchanged as if on unprotected system.

Dynamic Downgrading

There are three type of processes in our system: Downgradable benign processes, non-downgradable benign processes and untrusted processes. They are different based on the userids they have. Non-downgradable benign processes and untrusted processes in our system are characterized by having the same value for all three of their uids. Downgradable benign processes have both the ruid and euid set to that of the real user, with suid corresponding to an untrusted user. This factor enables downgradable processes to change their label from *benign* to *untrusted* at *any time during their execution*. Naturally, care needs to be exercised to ensure that this switch operation is secure.

While downgrading a process in our system is as simple as calling `setresuid` with the untrusted userid as a parameter. However, simply downgrading in this manner won't be secure because a downgraded process may file descriptors that were opened when the process was benign. As in the case of execve-time downgrade, open file descriptors need to be examined. However, instead of closing these file descriptors, as was done in the case of `uudo`, we prefer the simpler option of failing the `setresuid` operation if there are open file descriptors that enable the process to write to a benign file. This simple option is compatible with the dynamic downgrading primitive needed in our system. (See Section 5.5)

For a single process, it is easy to check its open file descriptors to determine if any of them point to benign files. But things become more complex in the presence of IPC. When multiple processes are connected via IPC, information can flow from one process to another. If any of the downstream processes have a high integrity file opened for writing, upstream processes should not be allowed to open a low integrity file for reading.

When multiple processes are connected via IPC, our system creates a shared memory region. Each of the benign member of the connected processes shares its opened file information with other processes. The shared memory region contains information for each of the interacting processes, and this information states if the process has a high integrity file opened for writing. This information is sufficient to determine whether a downgrade should be allowed to proceed or not. If so, the process can mark itself as downgraded, detach itself from the shared memory, and downgrades itself. In addition, processes keep monitoring the integrity of upstream processes, and if any of them become low integrity, they will downgrade themselves as well.

Apart from userid, processes can also access resources based on groupid and supplementary groups. groupid can be downgraded similar to how userid is downgraded. However, there is no mechanism to support downgrading supplementary groupids. As such, supplementary group accesses are no longer supported in the system. For supplementary group accesses, the system convert it to helper-based access as in Spif for downgradable processes. Downgradable processes are therefore started with no supplementary groups.

Limitations

One of the obvious limitation of the user-level dynamic downgrading approach is that a process can only downgrade at most once during its execution. This is because at most two userids can be propagated across exec, and every downgrading would consume one userid. However, a new userid can be resupplied by exec. The system can therefore allows processes to downgrade once during its execution.

Another limitation is that the level to downgrade to have to be decided ahead of time. This restriction, again, stems from the fact that processes can only carry two userids across exec.

4.5 Implementation

We implemented our dual-sandboxing enforcement mechanism on Ubuntu, PCBSD, and Windows. Our primary implementation was performed mainly on Ubuntu 10.04 and Windows 8. We ported our system to PCBSD, one of the best known desktop versions of BSD, to illustrate its feasibility on BSD system. In addition, we also tested the system on Windows XP, 7, 8, and Windows 10. We discuss the implementation specifics below:

4.5.1 Initialization

When our system is installed, existing files are considered as benign. Permission on existing world-writable files need to be changed so that they are not writable by untrusted processes.

Ubuntu We found no world-writable regular files on Ubuntu, so no permission changes were needed. There were 26 world-writable devices, but our approach did not change their permissions because they do not behave like files. Our approach also left permissions on sockets unchanged because some BSD systems ignore permissions on sockets. Instead, our system performs checking during the `accept` system call. World-writable directory with sticky-bit set were left unmodified because OSes enforce a policy that closely matches our requirement. Half of the 48 world-executable setuid programs were modified to group-executable by G_B. The rest were setgid programs and were protected using ACLs.

Windows Installation of our approach on Windows involves modifying ACLs. As Windows ACL supports inheritance, only a few directories and registry entries need updating— by default, Windows allows any user to create files in the root directory (e.g., `C:\`) and various system directories. Our system modified ACL to protect these directories such that untrusted files can only be created in shadowed or redirected directories. Instead of creating a new group G_B on Windows, our mechanism created negative ACL entries to revoke the write permissions of these directories. It also granted read and traversal permissions to R_U on R's home directory and registry subtree. This removes the need of the helper process U_H.

Some applications (e.g., Photoshop) intentionally leave some directories and files as writable for everyone. As such, untrusted processes could also write to these locations. Our approach prevented untrusted processes from writing into these locations by revoking write permissions from untrusted users. This was achieved by explicitly denying writes in ACLs. Once our system is installed, our system's benign sandbox will automatically modify ACLs to newly created world-writable files/directories/registry entries.

Some system files are writable by all users, yet they are protected by digital signatures. Our system currently does not consider digital signatures as provenance label, and hence it grants benign processes exceptions to read these "untrusted" files. A better approach is to incorporate signatures into provenance label so that no exception needs to be granted.

Apart from files, there were also other world-writable resources such as named pipes and devices for system-wide services. Our system granted exceptions for these resources as none of them could be controlled by untrusted processes, and these resources do not carry untrusted information.

Our system also created a shadow and a redirect directory and granted untrusted processes full control to the directories.

U_L and B_L realization

Policy enforcement mechanics On Ubuntu, our system modifies the system library at the binary level to realize the functionality of U_L and B_L. Fifteen assembly instructions were inserted around each system call invocation sites in system libraries (`libc` and `libpthread`). This allows our system to intercept all system calls. Our implementation then modifies the behavior of these system calls as needed to realize the sandboxes described in Section 4.2 and 4.3. Our system also modified the loader to refuse loading untrusted libraries for benign processes.

We cannot rely on the same mechanism to hook on Windows APIs. This is because Windows protects DLLs from tampering using digital signatures. Instead, our approach relies on the dynamic binary instrumentation tool *Detours* [Microsoft Research, 2015]. Detours works by rewriting in-memory function entry-points with jumps to specified wrappers. Our system builds wrappers around low-level APIs in `ntdll.dll` to modify API behaviors.

To initiate API-hooking, our approach injects U_L and B_L into every process. Upon injection, the `DLLMain` routines of U_L and B_L will be invoked, which, in turn, invoke Detours and trigger the API interception.

Our approach relies on two methods to inject U_L and B_L into process memory. The first one is based on `AppInit_DLLs` [Microsoft, 2015d], which is a registry entry used by `user32.dll`. Whenever `user32.dll` is loaded into a process, the DLL paths specified in the registry `AppInit_DLLs` will also be

	Shared	Ubuntu	PCBSD
Require no instrumentation	118	170	205
Benign Sandbox	49	6	29
Untrusted Sandbox	55	7	40

Figure 4.3: Number of system calls intercepted in Ubuntu and PCBSD

loaded.

A second method is used for a few console-based applications (e.g., the SPEC benchmark) that don't load `user32.dll`. This method relies on the ability to create a child process in suspended state (by setting the flag `CREATE_SUSPENDED`). The parent then writes the path of the dll into the memory of the child process, and creates a remote thread to run `LoadLibraryA` with this path as argument. After this step, the parent releases the child from suspension.

We rely on the first method to bootstrap the API interception process. Once the library is loaded into a process, all descendants of the process will be intercepted by making use of the second method. Although our approach may miss some processes started at the early booting stage, most processes (such as the login and Windows Explorer) are intercepted.

Enforcement on system calls Figure 4.3 shows the number of system calls that out approach instrumented to enforce policies on Ubuntu and PCBSD. On i386 Linux, some calls are multiplexed using a single system call number (e.g., `socketcall`). We demultiplexed them so that the results are comparable to BSD. Most of the system calls require no instrumentation. A large number of system calls that require instrumentation are shared between the OSes. Note that some calls, e.g., `open`, need to be instrumented in both sandboxes.

A large portion of the PCBSD specific system calls are never invoked: e.g., NFS, access control list, and mandatory access control related calls. Of those 59 (10 overlaps in both sandboxes) system calls that require instrumentation, 29 are in the benign sandbox. However, only 4 (`nmount`, `kldload`, `fexecve`, `eaccess`) out of the 29 calls are actually used in our system. Hence, we only handle these 4 calls. For the rest of the calls, we warn about the missing implementation if there is any invocation. The other 40 calls in untrusted sandbox are for providing transparency. We found that implementing only a subset of them (`futimes`, `lchmod`, `lutimes`) is sufficient for the OS and applications like `Firefox` and `OpenOffice` to run. Note that incomplete implementation in the transparency library U_L does not compromise security.

On Windows, our system intercepts mainly the low-level functions in `kernel32.dll` and `ntdll.dll`. Higher-level Windows functions such as `CreateFile(A/W)` , `CopyFile(A/W)`, `MoveFile(A/W)`, `ReplaceFile(A/W)`, `GetProfile...` `FindFirstFile(A/W)`,`FindFirstFileEx`, [8] rely on a few low-level functions such as `NtCreateFile`, `NtSetInformationFile` and `NtQueryAttributes`. By intercepting these low-level functions, all of the higher-level APIs can be handled. Our experience shows that changes to these lower level functions are very rare[9]. Moreover, some applications such as `cygwin` don't use higher-level Windows APIs, but still rely on the low-level APIs. By hooking at the lower-level API, our system can handle such applications as well. Figure 4.4 shows a list of API functions that our system intercepts.

There are 276 system calls in 32-bit Windows XP and 426 in 32-bit Windows 8.1. Although the number of system calls on Windows and Unix are comparable, system calls on Windows are more complicated than those on Unix. Windows has a large number of higher-level APIs that are translated into lower-level APIs by DLLs. For example, a file-open on Unix (`open(2)`) takes up to 3 arguments. On the other hand, file-open on Windows requires `NtCreateFile`, which takes 11 arguments. Apart from the file-path and open-mode, additional arguments are for setting file attributes, storing request completion status, setting allocation size, controlling share access, specifying how the file is accessed, and setting extended attributes. As such, the number of system calls that our approach needs to handle is much less than on Ubuntu. Apart from low-level APIs, our system also intercepts a few higher-level functions as they provide more context that enables policies to make better choices. For example, our system intercepts `CreateProcess(A/W)` to check if a benign executable is being passed an untrusted file argument, and if so, create an untrusted process. This has allowed policy inference Chapter 6.

[8] Calls ending with "A" are for ASCII arguments, "W" are for wide character string arguments.

[9] We did see new functions in Windows 8.1 that our system needed to handle.

API Type	APIs
File	`NtCreateFile`, `NtOpenFile`, `NtSetInformationFile`, `NtQueryAttributes`, `NtQueryAttributesFile`, `NtQueryDirectoryFile`,...
Process	`CreateProcess(A/W)`
Registry	`NtCreateKey`, `NtOpenKey`,`NtSetValueKey`, `NtQueryKey`, `NtQueryValueKey`,...

Figure 4.4: Windows API functions intercepted by dual-sandbox architecture

	LOC				
	C/C++		header		Other
	Ubuntu	+PCBSD	Ubuntu	+PCBSD	Both
Shared	2208	130	737	27	39
helper U_H	703	16	106		
uudo	68	52			
$B_L \cap U_L$	811	15	492	30	74
B_L only	451	67			
U_L only	944	81			
Total	5185	361	1335	57	113

Figure 4.5: Code complexity for realizing dual-sandbox on Ubuntu and PCBSD

4.5.2 Code complexity

Ubuntu and PCBSD Figure 4.5 shows the code size of different components for supporting Ubuntu, and the additional code for PCBSD. The overall size of code is not very large. Moreover, a significant fraction of the code is targeted at application transparency. We estimate that the code that is truly relevant for security is less than half of that shown, and hence the additions introduced to the TCB size are modest. At the same time, our system *reduces* the size of the TCB by a much larger amount, because many programs that needed to be trusted to be free of vulnerabilities don't have to be trusted any more.

Windows As for Windows, our system consists of 4000 lines of C++ and 1500 lines of header. This small size is a testament to the design choices made in our design. In particular, the helper U_H is removed because of the ACL support on Windows allows our approach to grant untrusted processes the precious set of permissions that U_H needs to grant. A small code size usually translates to a higher level of assurance about safety and security.

Part II

Policies

Chapter 5

Comparing Integrity Policies: Functionality, Compatibility, and Security

The second important component of our work is policy. A policy dictates how subjects and objects with different provenance labels can interact during an operation or within an execution. In this chapter, we consider several policies for integrity preservation. While all of these policies stop integrity violations, the manner in which they do so can differ substantially:

- **Stop (potential) violations by altering executions:** The most obvious resolution is to deny violating operations, as in the Biba [Biba, 1977] policy. However, simply denying the operation can lead to usability problems, e.g., a high integrity process attempting to run a low integrity program image would fail. Some policies allow this by downgrading the integrity of processes at runtime. Such downgrading, however, can create another usability problem: by the time the execution is known to be unsafe, there may no longer be an effective way to communicate the violation to the program. This leads to compatibility problems that can cause application crashes, which, in turn, may leave the system in an inconsistent state. ED/UII (Chapter 6) and SRFD (Section 5.5) are two new policies we explore to improve usability without impacting compatibility.

- **Rollback the changes made during executions:** The policies discussed above perform *eager enforcement:* they determine the safety of an operation before it is performed. However, there are scenarios where the safety of an operation cannot be determined until we examine subsequent operations. We discussed a *delayed enforcement* mechanism in the previous part of this dissertation in order to support such policies. With this mechanism in place, the effects of execution are isolated from the rest of the system, and they are subsequently examined to determine their safety. If this check succeeds, the effects will be committed and made visible to other processes in the system. Otherwise, the changes can be discarded as if the execution never occurred. The two advantages of delayed enforcement are that:

– it supports strictly more powerful policies as it allows intermediate unsafe changes, and

– it stops only those violating executions without needing to be overly conservative.

Note that eager enforcement is limited to the class of *enforceable policies* [Schneider, 2000], while deferred enforcement can support policies expressible using *edit automata* [Ligatti et al., 2005].

In this chapter, we discuss policies based on eager enforcement first, and then proceed to discuss policies that can be realized with delayed enforcement. Our goal is to compare these policies using three criteria: functionality, application compatibility, and user experience.

5.1 Criteria for usable policies

- **Functionality** concerns the ability of a policy to permit safe executions. Safe executions do not compromise the security goal of the systems and can run perfectly on unprotected systems. Ideally, a usable policy should permit every safe execution so that application actions are denied only when absolutely essential to preserve security. Permitting every safe execution, however, could lead to application compatibility problem below.

- **Application compatibility** concerns the ability of applications to recover from permission denials. One way to stop unsafe executions is to deny the operations that would compromise the security goal, e.g., a write operation by a low integrity process to a high integrity file. However, denying the operations at such a late stage could lead to compatibility problems: Programs may not be designed to handle such failures and may not have any recovery mechanism to recover from inconsistent states.

  Leaving systems in an inconsistent state also affects usability. One of the criteria of policies is to fail unsafe executions gracefully rather than abruptly. Policies use different strategies to detect potential unsafe executions and fail them earlier in the executions (promote early failures) as to achieve application compatibility.

- **User experience** concerns additional user prompts, the need for users to make security decisions or to modify their behavior. Ideally, security mechanisms should not assume that users are well-informed about the security consequences of their actions, or that they will take the time to carefully and thoughtfully answer security prompts.

In order to define these criteria formally, we begin by formalizing process execution in terms of the sequence of actions $\mathbf{A} = A_1 \ldots A_n$ performed by it[1]. Each action A_i can be:

- *an invocation (I),* typically, the execution of another program;
- *an observation (O),* typically, a file read operation; or
- *a modification (W),* typically, a file write operation.

In order to simplify terminology and description, we consider only two integrity levels in this chapter: benign/high (Hi) and untrusted/low (Lo). Objects (typically files) as well as subjects (processes) have one of these integrity levels.

[1] Schneider [Schneider, 2000] formulates enforceable security policies using the formalism of security automata. These automata make transitions that are entirely based on a subject's own operations, such as open's, read's and write's. Whereas these automata can only accept or reject an execution sequence, Ligatti et al [Ligatti et al., 2005] proposed a more powerful automata called *edit automata* that could also suppress or modify a subject's actions. We also use automata to compare different downgrading schemes for information flow systems, but the transitions in our automata are not only dependent on the subject's actions, but also the state of the file system. This is because whether an operation opens a high or low-integrity file is a function of the file system state. Indeed, Ligatti et al [Ligatti et al., 2005] explicitly specify that security properties in their model are those that are purely functions of the operation sequence.

Definition 6 (Integrity-preserving executions)
Integrity-preserving executions ensure that the content of all high integrity objects are derived entirely from other high integrity objects and subjects.

A strong integrity-preservation policy, such as the Biba's strict integrity policy and the low-water-mark policy, will ensure that all executions are integrity preserving. In particular, this means that low-integrity data and programs cannot influence the contents of integrity-critical (data or program) files on the system. Today's remote exploits and malware attacks all rely on modifying critical files using data or code from untrusted sources, and hence can be definitively blocked by enforcing these integrity policies, provided that only data and code from trustworthy sources is given a high integrity label.

A security policy can alter an execution sequence in one of two ways. First, it can disallow an operation A_i, denoted as $\not{A}_i$. There are several possibilities here, including (a) silent suppression of A_i, (b) suppressing A_i and returning an error to the process performing this operation, and (c) replacing A_i with another allowable action. We primarily focus on the alternative (b).

A second avenue for the enforcement engine is to downgrade a subject before A_i, denoted $\downarrow A_i$. Note that such a downgrade may be an internal operation within a reference monitor enforcing the policy, and hence we may not explicitly show it in some instances.

We call executions without any failed operations *permitted* or *successful executions,* while the rest are called *failed executions.* The more execution sequences that a security policy permits, the less functionality will be lost as a result of security policy enforcement. This leads to the following definition comparing the functionality of different security policies.

Definition 7 (Functionality)
A security policy P_1 is said to be more functional than P_2, denoted $P_1 \supseteq_F P_2$, if and only if every execution sequence permitted by P_2 is also permitted by P_1.

Note that functionality defines a partial order on security policies, and hence two policies could be incomparable in terms of functionality. By permitting more executions, a more functional policy would seem to have weaker security than a less functional policy, thus capturing the tension between functionality and security.

5.2 Integrity policies

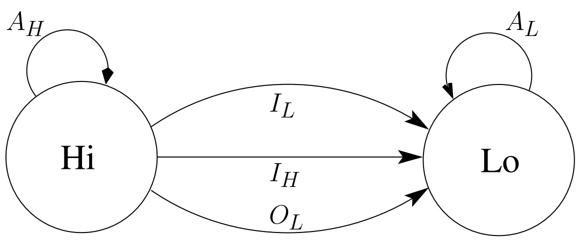
We can now classify actions into two categories: *high integrity actions* (A_H) that can be performed by high integrity subjects, and *low integrity actions* (A_L) that can be performed by low integrity subjects. Specifically, A_H includes all actions except read-down (O_L), i.e., read from a low-integrity input, and invoke-down (I_L), i.e., executing a program that has low integrity. A_L includes all actions except write-up (W_H). Note that I_H is permitted in A_L because we interpret it as the execution of a high integrity file within a low integrity subject. (In contrast, the term "invoke-up" is used in Biba model to refer to the execution of a high integrity subject.)

Integrity-preserving execution sequences can be realized by confining high integrity processes to perform only A_H, and low integrity processes to perform only A_L. Since W_H exists only in A_H and O_L exists only in A_L, it is clear that low-integrity objects and subjects cannot affect high-integrity objects.

Most systems do not allow revisions to object integrity levels. However, subject integrity label can be revised down as long as the downgraded subject is restricted to perform A_L after the downgrade. This leads to the following variants that all preserve integrity.

No Downgrading (ND). This policy, which corresponds to the Biba strict policy, permits no privilege revision (NPR): labels are statically assigned to subjects and objects, and they cannot change. With this strict interpretation, every program has to be labeled as high or low integrity, and a high integrity program cannot be used to process low integrity data, even if all outputs resulting from such use flow only to low-integrity files or low-integrity subjects.

Early downgrading (ED). This policy permits subject labels to be downgraded, but only when executing a program. This approach, also called *privilege revision on invocation* (PRI), allows more executions as compared to the no-downgrading policy. With the PRI policy, a subject wishing to operate on low-integrity files should know ahead of time (i.e., prior to execution) that it needs to consume low-integrity file, and drop its privilege before execution. This is why we call it *early downgrading.*



$A_H = \{O_H, W_H, W_L, I_H\}$
$A_L = \{O_H, O_L, W_L, I_L\}$

Figure 5.1: State machine for integrity-preserving executions.

Lazy downgrading (LD). The final policy is the low-water-mark policy for subjects, where downgrades can happen before any observe operation or an invoke operation. We call it lazy (or just-in-time) downgrading since downgrading operation would typically be delayed until the very last step, which must be the consumption of a low-integrity input.

Figure 5.1 shows a simple state-machine model that captures the above three policies. With the ND policy, none of the transitions between Hi and Lo states are available. With the ED policy, only the transitions on I_L and I_H are enabled. Note that while it is mandatory to transition to Lo on I_L, I_H may or may not cause a transition to Lo. When it does, it corresponds to the use of a high-integrity application to process low-integrity data.

With the LD policy, only the I_L and O_L transitions from Hi to Lo are enabled. There is no need to make a transition from Hi to Lo on I_H, as the downgrade can be deferred until the next operation to read low-integrity data. As a result, LD avoids one of the difficulties of ED, namely, the need to predict ahead of time whether a certain process will need to read low-integrity data. `uudo` inference described previously is a technique to make ED more usable.

5.3 Comparing functionalities of integrity policies

It is easy to see the motivation for the LD policy: when the actions performed by an application are disallowed, it can lead to errors and failures, and hence loss of functionality. In contrast, downgrading has the potential to permit the application to continue and function. In fact, we can formally state:

Theorem 8 $LD \supset_F ED \supset_F ND$

Proof: This theorem simply states that LD is strictly more functional than ED, which, in turn, is more functional than ND. From the definition of the three policies, and Figure 5.1, it is easy to see that all three policies accept the same set A_L^* of execution sequences for low-integrity subjects. (We are using a regular expression syntax to succinctly capture the set of execution sequences permitted by a policy.) Thus, we can limit our comparisons to the execution sequences permitted for high-integrity subjects. Note that ND accepts only sequences of the form A_H^* for high-integrity subjects. ED accepts $(I_L|I_H)A_L^*$ for subjects started with high, in addition to the set A_H^*. Finally, LD accepts $A_H^*(I_L|O_L)A_L^*$, which is a strict superset of sequences accepted by ED. ■

5.4 Compatibility

Increased functionality does not always translate to a better user experience, or better compatibility with existing software. Self-revocation is a prime example of the compatibility problem posed by LD, an approach that maximizes functionality over other integrity policies. In contrast, ED is less functional as compared to LD, but is considered more compatible.

Self-revocation occurs when a subject is initially granted access to a resource, but this access is revoked subsequently; and the revocation is the result of some of the other actions performed by the subject itself. More concretely, self-revocation manifests itself as follows in the context of file system APIs provided by modern operating systems: a process successfully opens a file, but a subsequent write operation using that file handle is denied. Although self-revocation is more commonly identified with failures of writes, it can also happen on read operations. In both cases, self-revocation raises several compatibility issues:

- The file system API is designed to perform security checks on open operations, but not on reads and writes. As a result, there is usually no way to even communicate a security failure to the subject performing the read or write[2]. Thus, security failures have to be mapped into other

[2]For instance, on UNIX, there are no error codes related to permissions that can be returned by read and write system calls.

failures that can occur on reads/writes, such as an attempt to read a file before opening it. Such remapping has obvious drawbacks because applications may misinterpret the error code and respond inappropriately.

- Even if an error code is returned on reads and writes, many applications may not check them at all. This is because failures of these operations are rare and unexpected, so many applications may not contain code for checking these error cases, or undertaking any meaningful error recovery.

- Even if the application checks the error and undertakes recovery, data loss or corruption may be unavoidable at this point. Consider an application that was updating a file. If its write access is taken away when it is half-way through the update, that may lead to the file being truncated, leading to data loss, inconsistency or corruption.[3]

For this reason, we develop the following notion of application compatibility, or simply, compatibility of security policies.

Definition 9 (Compatibility)
We say that a security policy P is compatible if all actions disallowed by it can return a valid permission failure error to the subject.

With contemporary file APIs, this means that a compatible policy would deny `open`'s but not reads/writes. We show that in terms of compatibility, the results are inverted from that of functionality:

Theorem 10 *LD is not failure-compatible, whereas ND and ED are both failure compatible.*

Proof: Recall that for subjects that start at low integrity, all three policies allow A_L^*. It is clear that this sequence permits the same set of operations throughout, so self-revocation is not possible. For high-integrity subjects, ND accepts A_H^* — again, the set of operations permitted remain constant throughout the subject's lifetime, and hence there will be no self-revocation. For ED, the sequences accepted are of the form A_H^* or A_L^*. For each alternate,

[3]With buffered I/O, even the data that an application believes to have written prior to the self-revoking action may be lost — such data may be held in the program's internal buffers, which may be flushed much later, at which point, the write system call would fail.

it is easy to see that all of the actions permitted towards the beginning of the sequence are also permitted later on, once again ruling out the possibility of self-revocation. Finally, we have already explained how *LD* suffers from self-revocation. ■

5.5 Self-Revocation Free Downgrading

LOMAC [Fraser, 2000] argues that a central reason for non-adoption of conventional information flow techniques is that of compatibility. They consider information flow systems that allow privilege revision (such as dynamic downgrades) and those that don't, and conclude that former class has increased compatibility. They point out that policies such as low-water-mark policy had not received much attention because of the self-revocation problem. They proceeded to address this problem in a particularly common case, namely, the pipelines created by shell processes. As noted earlier, their solution relied on the shell's use of Unix process groups to run each pipeline, and ensuring that all processes within such a group had identical integrity labels. In this manner, there will never be a need to restrict communications within a process group, and thus self-revocation involving pipes is prevented. They remark that they "cannot entirely remove this pathological case without also removing the protective properties of the model." Indeed, the solution they present does not attempt to address revocations involving files, sockets, etc. Our work is inspired by their comments, and shows that it is in fact possible to retain the security benefits of integrity protection, as well as the compatibility benefits of privilege revision without incurring the cost of self-revocation.

The results in the previous section lead to the following question: can there be an approach that is preferable in terms of both functionality and compatibility? Our answer is affirmative. We begin by positing the existence of a new dynamic downgrading policy that combines *LD*'s functionality with the compatibility of *ED*.

Definition 11 (Self-revocation-free downgrading)
SRFD accepts the same set of execution sequences as LD. Every sequence that is modified by LD is also modified by SRFD, but unlike LD, SRFD only modifies (i.e., denies) open operations.

So, the next natural question is whether SRFD is realizable. Conceptually, we can synthesize execution sequences accepted or modified by SRFD from the acceptance or modification actions of LD as follows. If LD accepts a sequence, then SRFD will accept the same sequence. If LD modifies a sequence, let A_i be the first write operation denied by LD. SRFD will identify the open operation A_j preceding A_i that caused LD to downgrade the subject, and then SRFD will deny A_j.

Noting that LD denies only write operations on high-integrity files, this means that SRFD needs to predict whether a subject will perform future writes on any of the currently open file descriptors for accessing high-integrity files. If so, SRFD should not permit the subject to open any low-integrity file. In this manner, SRFD can prevent the subject from downgrading itself, and hence will not have to deny writes on one of these descriptors in the future.

This raises the final question: how can a reference monitor predict future actions of a subject? Often, questions regarding future behavior are answered by assuming that any thing that can happen will indeed happen. We formalize this by characterizing a class of programs that transfer data along every possible communication channel between communicating processes, and show that for this class, SRFD can indeed be realized.

Another way to characterize our result is as follows. Unless an oracle for predicting future behavior of a set of communicating processes exists, one cannot improve over the functionality of the design presented in the next section without risking self-revocation.

Our implementation of SRFD on contemporary operating systems was based on the kernel-based mechanism described in Chapter 3. We show that SRFD eliminates self-revocations and unless future inter-process communications can be predicted accurately, it is not possible to improve on the functionality of SRFD without incurring self-revocation.

Theorem 12 *There will be no self-revocations experienced by the implementation described above.*

Proof: The proof is by contradiction. Suppose that a self-revocation takes place on a `read` or `write` operation that transfers data from A to B. From Observation 3, self-revocation can happen only when `current_lbl`$(A) <$ `min_lbl`(B). Together with Invariant 2, this implies that `min_lbl`$(A) <$ `min_lbl`(B). However, note that it is invalid to issue a read or write operation before setting up the information flow path between A and B. (In this case, the path happens to be of length 1.) From Invariant 4, the condition $\texttt{min\_lbl}(A) \geq \texttt{min\_lbl}(B)$ must also hold, thus leading to a contradiction. ■

Definition 13 (Flow-indeterminate programs) *A set of programs are said to be flow-indeterminate if for any set of communicating processes running them, the following condition holds: for every communication path p between any two processes, there will be data transfer operations that cause data to flow from the beginning to the end of this path.*

Flow-indeterminacy simply formalizes the idea that programs may exhibit any possible pattern of communication that is consistent with their current set of open file descriptors; and that there is no simple yet general way to delineate likely communications from those that are unlikely/impossible.

Theorem 14 *For flow-indeterminate programs, any policy that accepts any execution rejected by SRFD will suffer from self-revocation.*

Proof: For an execution sequence rejected by our approach, consider the first operation A_i that is denied. From the description of the approach in Section 3.3.3, A_i must be an `open` operation that would have created a path from entity A to B such that $\texttt{current\_lbl}(A) < \texttt{min\_lbl}(B)$. Now, suppose that there exists a correct integrity policy P that permits this `open` operation. Then, because of the properties of flow-indeterminate programs, it can be seen that there will be a subsequent operation that transfers data from A to B. This will either have to be denied, or it will cause $\texttt{current\_lbl}(B)$ to fall below $\texttt{min\_lbl}(B)$. The former case corresponds to self-revocation, thus completing the proof. In the latter case, from Observation 5, it can be seen that there is some output file B_i whose `min_lbl` is higher than $\texttt{current\_lbl}(B)$. Also, from properties of flow-indeterminate programs, there will be an actual data flow from B to B_i, which will cause the output file B_i's label to fall below its minimum value. This is not permissible in the model, and hence the more permissive policy P is simply invalid. Thus, in either case, we have established that the functionality offered by SRFD cannot be increased without risking self-revocation. ■

Thus, for flow-indeterminate programs, we have shown that SRFD allows the same successful executions as any other valid information-flow policy that is free of self-revocations. Thus, SRFD represents the maximal functionality achievable without any self-revocations. We present our implementation of SRFD in Chapter 9.

5.6 Integrity and Availability Guarantees

In this section, we provide a formal treatment of the integrity and availability guarantees that can be provided by the integrity preservation policies described in previous sections. Key difficulties in this formalization are those of defining what it means for a system to possess high integrity, and how to account for the effects of actions performed by malicious or careless users that may indeed compromise security. The novelty of our approach is that it enables these problems to be side-stepped.

In our analysis, we assume that the OS kernel is not vulnerable. We also assume that all files that kernel reads directly (not including those read on behalf of user-level processes) are writable only by the system administrator[4]. Since our system does not permit low-integrity code to run with system administrator privilege, the kernel will never read a low-integrity file, nor will it fail due to the absence of such a file (as low-integrity code could not have created the file in the first place). As a result, integrity and availability failures can only be experienced by user-level processes. Hence the proofs below target only at user-level processes.

We also assume that no low-integrity file will be marked high-integrity, regardless of how the file entered the system.

5.6.1 Integrity Preservation

We use $\mathbf{F}$ to denote the state of the file system. The file system state evolves discretely over time, with each change corresponding to the execution of a command c, denoted $\mathbf{F} \xrightarrow{c} \mathbf{F}'$. A command includes a program name, and encompasses both command-line arguments and environment variables. We denote a command invoked by a high-integrity process as high-integrity command. During an execution, a process may read a set of input files, and produce a set of output files. For simplification, we assume that commands are deterministic, i.e., two executions of the same program with identical command-line arguments, environment, and system state will result in exactly the same set of changes to $\mathbf{F}$. Some times we use the notation $\mathbf{F}(t)$ to denote system state at time t.

The most natural steps towards a proof would seem to be: (1) provide

[4]Recall from the description of U_L that this property holds for loadable kernel modules as well.

a formal definition of integrity, and (2) prove that our sandboxes preserve integrity after every command execution. Unfortunately, both steps are problematic. Firstly, contemporary OSes are too complex and dynamic to develop a precise yet practical definition of integrity. Secondly, even in the absence of any low-integrity code, system integrity could be compromised due to factors such as human errors, e.g., running an important system script with incorrect parameters. Thus, a formal proof requires a different way of thinking about the problem.

To overcome the first problem, we develop an abstract characterization of integrity: our proof relies on the existence of a function I for determining integrity, but does not require its details to be spelt out. Formally:

$$I : \mathbf{F} \times N \longrightarrow \{0, 1\}$$

such that system integrity held at some time t_0 (i.e., $I(\mathbf{F}(t_0), N) = 1$). Here, N is a subset of filenames in $\mathbf{F}(t_0)$ that are relevant for integrity. Moreover, we assume that all files in N are labeled as high-integrity at the start time t_0. In the degenerate case, where some of the files in N are not present in $\mathbf{F}$, I returns 0.

To overcome the second problem, i.e., side-step the effect of human errors, we don't prove that command executions *always* preserve integrity. Instead, we show that (a) if integrity is lost, then its root cause can be traced to a high-integrity command execution, and (b) the loss would have been experienced even if there was no low-integrity code or data on the system.

Lemma 15 *In the absence of trusted processes (i.e., processes that remain high-integrity after consuming low-integrity inputs), system integrity cannot be compromised due to the presence of low-integrity files.*

Proof: If integrity is never lost, then there is nothing to prove. Otherwise, let t_m be the earliest time when $I(\mathbf{F}(t_m), N) = 0$. Let us denote the evolution of system state from t_0 to t_m as follows:

$$\mathbf{F}(t_0) \xrightarrow{c_1} \mathbf{F}(t_1) \xrightarrow{c_2} \mathbf{F}(t_2) \xrightarrow{c_3} \cdots \xrightarrow{c_m} \mathbf{F}(t_m) \tag{5.1}$$

From this sequence, we construct another sequence

$$\mathbf{F}_{Hi}(t_0) \xrightarrow{c'_1} \mathbf{F}_{Hi}(t_1) \xrightarrow{c'_2} \cdots \xrightarrow{c'_m} \mathbf{F}_{Hi}(t_m) \tag{5.2}$$

that consists of only high-integrity commands and operates on the restriction $\mathbf{F}_{Hi}$ of the system state to files that are labeled as high-integrity. Command c'_i is the null command (i.e., it leaves the system state unchanged) if c_i represents a low-integrity command, or else $c'_i = c_i$. We now establish the validity of Sequence (5.2) by induction on m. The base case of $m = 0$ holds vacuously. For the induction step, assume that the sequence is valid up to length $k-1$. In the k^{th} step, if c_k is a low-integrity process, then, the policies enforced by the sandboxes ensure that it cannot modify any high-integrity files. Thus, $\mathbf{F}_{Hi}(t_{k-1}) = \mathbf{F}_{Hi}(t_k)$, thus validating the k^{th} step with c_k being the null command. If c_k is a high-integrity process, then, because of the fact that none of the high-integrity processes are permitted to consume low-integrity inputs (or even query their existence), the behavior of c_k is solely determined by the content of high-integrity files in $\mathbf{F}(t_{k-1})$. Due to determinism of command execution, c_k will have identical effects on the file system when executed on $\mathbf{F}(t_{k-1})$ and $\mathbf{F}_{Hi}(t_{k-1})$, so $\mathbf{F}_{Hi}(t_{k-1}) \xrightarrow{c_k} \mathbf{F}_{Hi}(t_k)$ in this case too.

Since t_m is the first step where integrity does not hold, all of the files in the set N are present in $\mathbf{F}(t_k)$ for $0 \leq k < m$. Moreover, due to our policies that do not permit overwriting a high-integrity file with a low-integrity one, the files in N will always be high-integrity. This means that $\mathbf{F}_{Hi}(t_k)$ will include all of the files in N, and hence $I(\mathbf{F}_{Hi}(t_k), N) = 1$ for $0 \leq k < m$. Since we assumed $I(\mathbf{F}(t_m), N) = 0$, either some of the files in N are not present in $\mathbf{F}(t_m)$, or I returns 0 on these files. In either case, $I(\mathbf{F}_{Hi}(t_m), N) = 0$. Thus, integrity violation occurs in the Sequence (5.2) that has no low-integrity executions or files. ■

Theorem 16 *Sandboxes U_I, U_L and U_H ensure that system integrity is not compromised by low-integrity applications.*

Proof:

The proof of Lemma 15 amounted to showing that if there was an integrity violation, it was due to high-integrity processes. In this theorem, we have one more reason for integrity violation, namely, a (buggy) trusted process that compromises the system due to consumption of low-integrity input. This error should be attributed to the trusted process. We formalize this idea by treating the input as if it was already embedded in the code of the trusted process, and then removing the external input.

Specifically, as in the proof of Lemma 15, we start with the sequence that violates system integrity properties, i.e.,

$$\mathbf{F}(t_0) \xrightarrow{c_1} \mathbf{F}(t_1) \xrightarrow{c_2} \mathbf{F}(t_2) \xrightarrow{c_3} \cdots \xrightarrow{c_m} \mathbf{F}(t_m) \tag{5.3}$$

From this sequence, for each trusted process execution c_k in this sequence that safely consumes a low-integrity file f, we construct another command $c_{k,f}$ that has the exact same behavior, except that it does not read f at all:

$$\mathbf{F}(t_0) \xrightarrow{c_1} \mathbf{F}(t_1) \xrightarrow{c_2} \mathbf{F}(t_2) \xrightarrow{c_3} \cdots \mathbf{F}(t_{k-1}) \xrightarrow{c_{k,f}} \cdots \xrightarrow{c_m} \mathbf{F}(t_m) \qquad (5.4)$$

This transformation has the effect of attributing integrity violations due to the consumption of f as an error that is contained entirely within the trusted process. Once this is done, we have a sequence that is the same as at the beginning of the proof of Lemma 15, i.e., a sequence without considering trusted programs. We can reuse that proof to establish that low-integrity processes and files were not the source of any integrity violations.

■

5.6.2 Availability Preservation

Although the sandboxing of high-integrity processes has the benefit of enhancing system integrity, availability may be degraded since these processes may now fail due to accesses being denied. However, we show that this cannot happen in our system.

We use a construction similar to the proof of Theorem 16 to show that any availability loss is due to high-integrity commands only. We define a function A for determining availability similar to I:

$$A : \mathbf{F} \times N \longrightarrow \{0, 1\}$$

Here, N is a subset of filenames in $\mathbf{F}(\mathbf{t_0})$ that are relevant for availability. We assume that files in N can be partitioned into two sets: N_I contains the set of high-integrity at the start time t_0, and N_O contains the set of optional files that can either be of high-integrity or do not exist. Non-existence of files in N_O (e.g., preference files) do not compromise the availability of applications because applications can create them in the first run.

Similar to the proof for integrity, we don't prove that benign command executions always preserve availability. Instead, we show that (a) if availability is lost, then its root cause can be traced to a high-integrity command execution, and (b) the loss would have been experienced even if there was no low-integrity code or data on the system.

Lemma 17 *In the absence of trusted processes, system availability cannot be compromised due to the presence of low-integrity files.*

Proof: If availability is never lost, then there is nothing to prove. Otherwise, let t_m be the earliest time when $A(\mathbf{F}(t_m), N) = 0$. Let us denote the evolution of system state from t_0 to t_m as follows:

$$\mathbf{F}(t_0) \xrightarrow{c_1} \mathbf{F}(t_1) \xrightarrow{c_2} \mathbf{F}(t_2) \xrightarrow{c_3} \cdots \xrightarrow{c_m} \mathbf{F}(t_m) \tag{5.5}$$

From this sequence, we construct another sequence

$$\mathbf{F}_{Hi}(t_0) \xrightarrow{c'_1} \mathbf{F}_{Hi}(t_1) \xrightarrow{c'_2} \cdots \xrightarrow{c'_m} \mathbf{F}_{Hi}(t_m) \tag{5.6}$$

that consists of only high-integrity commands and operates on the restriction $\mathbf{F}_{Hi}$ of the system state to files that are labeled as high-integrity. Command c'_i is the null command (i.e., it leaves the system state unchanged) if c_i represents a low-integrity command, or else $c'_i = c_i$. We now establish the validity of Sequence (5.2) by induction on m. The base case of $m = 0$ holds vacuously. For the induction step, assume that the sequence is valid up to length $k - 1$. In the k^{th} step, if c_k is a low-integrity process, then, the policies enforced by the sandboxes ensure that it cannot modify any high-integrity files. Thus, $\mathbf{F}_{Hi}(t_{k-1}) = \mathbf{F}_{Hi}(t_k)$, thus validating the k^{th} step with c_k being the null command. If c_k is a high-integrity process, then, because of the fact that none of the high-integrity processes are permitted to consume low-integrity inputs or even query existence of low-integrity files, the behavior of c_k is solely determined by the existence and content of high-integrity files in $\mathbf{F}(t_{k-1})$. Due to determinism of command execution, c_k will have identical effects on the file system when executed on $\mathbf{F}(t_{k-1})$ and $\mathbf{F}_{Hi}(t_{k-1})$, so $\mathbf{F}_{Hi}(t_{k-1}) \xrightarrow{c_k} \mathbf{F}_{Hi}(t_k)$ in this case too.

Since t_m is the first step where availability does not hold, the following conditions hold for $0 \leq k < m$:

- all of the files in the set N_I are present in $\mathbf{F}(t_k)$, and
- all of the files in the set N_O are either present in $\mathbf{F}(t_k)$ or does not exist

Moreover, due to our policies that do not permit overwriting a high-integrity file with a low-integrity one, and high-integrity processes cannot determine the existence of low-integrity files, the files in N_I will always be high-integrity, and the files in N_O will always be either high-integrity or their existence cannot be determined by high-integrity processes. This means that $\mathbf{F}_{Hi}(t_k)$ will include all of the files in N. The sandbox U_I and hence $A(\mathbf{F}_{Hi}(t_k), N) = 1$ for $0 \leq k < m$. Since we assumed $A(\mathbf{F}(t_m), N) = 0$, either some of the files in N_I are not present in $\mathbf{F}(t_m)$, or A returns 0 on files in N_I or N_O. In either case, $A(\mathbf{F}_{Hi}(t_m), N) = 0$. Thus, availability violation occurs in the Sequence (5.6) that has no low-integrity executions or files. ■

Theorem 18 *Sandboxes U_I, U_L and U_H ensure that availability of high-integrity processes will not be compromised due to the presence of low-integrity files or their execution.*

Proof: The proof of Lemma 17 amounted to showing that if there was an availability failure, it was due to high-integrity processes. In this theorem, we have one more reason for availability failure, namely, a (buggy) trusted process that compromises the system availability due to consumption of low-integrity input. This error should be attributed to the trusted process. Similar to integrity, we formalize this idea by treating the input as if it was already embedded in the code of the trusted process, and then removing the external input.

Specifically, as in the proof of Lemma 15, we start with the sequence that violates availability properties. For each trusted process execution c_k in this sequence that safely consumes a low-integrity file f, we construct another command $c_{k,f}$ that has the exact same behavior, except that it does not read f at all. This transformation has the effect of attributing availability failure due to the consumption of f as an error that is contained entirely within the trusted process. Once this is done, we have a sequence that is the same as at the beginning of the proof of Lemma 17, so we can reuse that proof to establish that low-integrity processes and files were not the source of any availability failures.

Relaxing Assumptions in the Formal Model One of the assumptions was deterministic execution. This can be relaxed by permitting process executions to transform an input state into one of many possible states, with the provision that the possibilities remain identical for the same input state. This would allow the proof to be easily carried through.

A second assumption was sequential execution of processes. This can easily be relaxed to permit common cases where multiple processes interact with each other, while executing concurrently. For instance, consider the common case of a script that starts up several processes over its lifetime, and ensures that all of these processes are terminated before its own termination. This set of process executions can be captured as if it is one large command that executed.

5.7 Delayed Enforcement

In this section, we discuss policies that can be realized with *delayed enforcement* (DE), which requires rollback and commit capabilities from the enforcement framework. Unlike ED/UII and SRFD, DE is not based on downgrading. DE first isolates the effect of an execution. When the execution has completed, the effect of the execution will be checked by the policy. If the policy determines the changes are safe, DE will commit the effects. Otherwise, the changes will be discarded as if the execution never took place.

It is clear that delayed enforcement can permit all executions that can be determined to be safe by eager enforcement. In addition, DE can permit some executions that go through unsafe intermediate states before reaching a safe final state. Such executions would be denied by previously discussed integrity policies. For this reason, DE's functionality is strictly superior to that of ND, ED, LD, or SRFD. Moreover, using a rollback capability, it is possible to cleanly recover from aborted executions. In other words, it is no longer necessary to communicate security errors to a subject in a meaningful way: if an operation is to be denied, and there isn't a way to communicate permission denial to an application, the whole execution can be rolled back[5]. Thus, DE can support policies that provide better compatibility than all of the previously discussed integrity policies.

Finally, DE policies can be simpler to express because they can reference the state of the system at the end of an execution. In contrast, previous integrity policies needed to be stated in terms of permissibility of individual operations. We present a compelling illustration of this ability in Section 10.

DE provides semantics similar to the transactional semantics of edit automaton [Ligatti et al., 2005], which basically suppress all the actions of the execution, and emit the execution when the edit automaton determined that the entire execution is safe.

[5]Note that since applications are executed in isolation, rollbacks don't propagate outside of this isolated environment. At the end of execution, if the policy is satisfied, the results are committed at that point, and become visible to the processes that were executing outside of this isolated environment. See Reference [Liang et al., 2009] for how such an environment can be realized.

5.7.1 Discussion

The main reason for resolving policy violations is because they can lead to partial completion when executions failed due to the unexpected failures introduced by the security systems. Both *ED* and SRFD make sure all failures are compatible — fail at open so that applications may handle them more gracefully. As discussed previously, there is a trade-off between functionality and compatibility. A system cannot have both because one has to decide allow or deny for each action. An alternative approach is to build recovery mechanisms to "rollback" failed executions. This is not always easy to do in general.

One-way isolation [Sun et al., 2005] uses rollback as the default choice, while providing primitives to commit executions that the user determines to be secure. However, it is problematic to rely on users to decide what is secure. Not only does it demand considerable time, effort and skill on the part of users, but also suffers from the fact that users could be easily fooled. Thus, rollback techniques coupled with automated procedures for determining secure executions are needed. Such automated procedures require full specification of what is secure — this itself is too a difficult task to accomplish in general. However, it may be possible to specify detailed and accurate policies for secure execution in special cases. One example of this is the secure software installation [Sun et al., 2008a] work, where a policy for determining secure installations was specified and checked automatically.

TxBox [Jana et al., 2011] relies on TxOS [Porter et al., 2009] to sandbox and isolate untrusted processes using transactions. It allows untrusted processes to run until they trigger policy violations. A policy violation is defined at the system call level. When violations occur, the system state can be rolled back as if the untrusted process never ran.

Back to the future [Hsu et al., 2006] takes this "rollback" technique to the extreme. It protects system integrity against malware by rolling the system back. Instead of confining untrusted processes, it confines benign processes. Every modification made by untrusted processes is recorded. Whenever a benign process consumes untrusted data, the entire system will be rolled back to the "clean" state, and the untrusted process will be terminated. This allows high-integrity processes to consume only high-integrity data.

Chapter 6

Policy Inference

Although the policies described in the last chapter seemed fully defined, there arise several situations where multiple safe alternatives are available, and it is unclear which of those alternatives is more desirable. For instance, consider the early downgrade policy. There are situations where the correct option it to deny the execution of an untrusted executable. In other situations, the appropriate action is to downgrade the process and to allow the execution to proceed. One way to resolve this choice is to ask the users to explicitly indicate if they wish a downgrade to be performed. The *uudo* helper discussed in the previous chapter provides a mechanism for doing this. Unfortunately, it creates too much work for users if they have to manually identify downgrades every time.

Another situation where a choice arises is as follows. Suppose that an untrusted subject U tries to overwrite a benign object B. This action could simply be denied, but this choice may cause U to fail. A second alternative is to permit the U to create a shadow object B_S that is only visible to untrusted subjects. With this choice, subsequent references to B by untrusted subjects would be redirected to B_S, while benign subjects would directly access B. Once again, most users won't have the time and/or knowledge needed to determine which of the two choices (denial or shadowing) is appropriate.

In this chapter, we develop techniques for automatically inferring the right choice in scenarios such as those described above. While the inference techniques are not guaranteed to resolve among possible choices in all situations, they do so in most common cases, thereby minimizing user involvement in security decisions.

6.1 Policy inference for untrusted subjects

Our policy for untrusted processes is geared to stop actions that have a high likelihood of damaging benign processes. A benign process may be compromised by altering its code, configuration, preference, or data input files. Of these, the first three choices have a much higher likelihood of causing harm than the last as programs are less likely to be written to protect against them. For this reason, our policy for untrusted processes is based on denying access to code, configuration, and preference files of benign processes. However, note that benign applications may be run as untrusted processes, and in this case, they may fail if they aren't permitted to update their preference files. For this reason, our approach is to shadow writes to preference files, while denying writes to configuration and code files.

To implement this policy, we could require system administrator (or OS distributors) to specify code, configuration, and preference files for each application. But this is a tedious and error-prone task. Moreover, these details may change across different software versions, or simply due to differences in installation or other options.

A second alternative is to do away with this distinction between different types of files, and apply shadowing to all benign files that untrusted processes opened for writing, i.e., create an untrusted copy for every benign file. But this approach has several drawbacks as well:

- Shadowing should be applied to as few files as possible, as users are unaware of these files. In particular, if data files are shadowed, users may not be able to locate them. Thus, it is preferable to apply shadowing selectively to preference files. (Users can still find the redirected files as they are visible to benign processes.)

- If accesses to all benign files are shadowed, this will enable a malicious application to compromise *all* untrusted executions of benign applications. As a result, no benign application can be relied on to provide its intended function in untrusted executions. (Benign executions are not compromised.)

- Finally, it is helpful to identify and flag accesses that are potentially indicative of malware. This helps prompt detection and/or removal of malware from the system.

We therefore develop an automated approach for inferring different categories of files, and then apply shadowing to a narrow subset of files.

6.1.1 Explicitly specified versus implicit access to files

When an application accesses a file f, if this access was triggered because of external input, then this access is considered to be explicitly specified. For instance, f may be specified as a command-line argument or as an environment variable. Alternatively, f may have been selected by a user using a file selection widget. All file accesses that are *not* explicit are deemed implicit.

Applications seldom rely on an explicit specification of their code, configuration, and preference files. Libraries required are identified and loaded automatically without a need for listing them by users. Similarly, applications tend to "know" their configuration and preference files without requiring user input. In contrast, data files are typically specified explicitly. Based on this observation, we devise an approach to infer implicit accesses made by *benign* applications. We monitor these accesses continuously and maintain a database of implicitly accessed files, together with the mode of access (i.e., read-only or read/write) for each executable. The policy for untrusted subjects is developed from this information, as shown in Figure 6.1.

Note that our inference is based on accesses of benign processes. Untrusted executions (even of benign applications) are not considered, thus avoiding attacks on the inference procedure.

6.1.2 Computing Implicitly Accessed Files

Files that are implicitly accessed by an application are identified by exclusion: they are the set of files accessed by the application but are not explicitly specified. Identifying explicitly specified files can be posed as a taint-tracking problem. Taint sources include:

- command-line parameters
- all environment variables
- file names returned by a file selection widget, which captures file names selected by a user from a file dialog box

	Implicitly accessed by benign		Explicitly accessed
	read and write	other	
Inferred type	Preference	Code and configuration	Data
Action	Shadow	Deny	Deny

Figure 6.1: Untrusted Sandbox policy on modifying benign files

Taint in our system is propagated with the following rule: If a directory with a tainted file name is opened, all of the file names from this directory are marked as tainted. This is important for file selection dialogs: when users open a directory by double-clicking on a directory icon in file selection dialogs, this directory open should be regarded as explicit because users are involved. Hence, intermediate paths returned should also be intercepted. Explicitly specified files are those that are tainted.

Our system intercepts `exec` and `CreateProcess` to monitor arguments and environment variables. It also intercepts values returned by file selection widgets to capture what files users selected. These values are then used to determine if a file access is explicit. We consider a file access as explicit if the file accessed has a name that matches the explicit values specified by the users. Other files are regarded as implicitly accessed.

In terms of implementation, the file name for each file open is matched against a set of explicit values. Furthermore, values specified by users may not be exactly the same as the file name appears in `open`(2): a file may be specified via command line options: e.g., $-$`config` $=$ `test`. The file eventually opened can be `/path/to/file/test.cfg`. The location of the file may be the current working directory of the process, or a default directory specific for the application. The argument contains program specific option `config`, followed by `test`, which appears in the actual open system call argument. We rely on the assumption that file names typically do not contain "−". If it identifies an argument starts with "−" or "=," then it discards parts of the name up to these symbols. Instead of performing an exact match, we consider matches if the length of the longest substring between the file accessed and any of the explicitly specified values exceeds certain size. A match will be added to the set of explicit values. our approach is to applies Aho-Corasick [Aho and Corasick, 1975] algorithm to compute the longest common substring and supports incremental tracking of explicit values efficiently.

	Implicit access	Explicit access
Action	Deny (file does not exist)	Deny (permission denied unless trust-confined)

Figure 6.2: Benign Sandbox policy on reading untrusted files

With information on whether a file is accessed implicitly by programs or explicitly by users, different policies can be applied to serve users better. This "implicit-explicit" technique not only can be used to infer file types and hence be used for shadowing policy, but also be applied to limit the trust on programs that need to handle benign and untrusted simultaneously (Section 6.2).

6.2 Policy inference for benign subjects

Policies can also be inferred for benign programs. We discuss how we automate the policy inference process for each of the three modes supported on benign code.

6.2.1 Logical isolation

The default policy for benign code is to prevent consumption of untrusted inputs, while returning a "permission denied" or "file does not exist" error code.

Although an attempt to open a file is always going to be blocked, the response of the application can be different, based on the error code returned. We find that applications handle "permission denied" error codes well when dealing with data files, i.e., user specified files. Moreover, communicating this error message to the user is less confusing than the alternative "file does not exist." Finally, it can suggest to the user to run the application with the `uudo` wrapper. On the other hand, some applications (e.g., OpenOffice on Windows) are less graceful in handling permission denials on configuration and preference files. This is because applications create these configuration and preference files and do not expect users to modify them. So, our system returns a "file does not exist" error when accessing untrusted files implicitly (Figure 6.2).

6.2.2 Untrusted execution

By design, ED does not allow subjects to change their integrity labels. Processes therefore have to decide which integrity level they want to be in before executing program images. To run as low-integrity, users can invoke `uudo`. Requiring users to explicitly invoke `uudo` has the benefit that users know in advance whether they can trust the outputs or not. However, it is an inconvenience for users to make this decision all the time. Hence, ED/UII can also automatically infer the use of `uudo`. The idea is as follows: if an execution will fail without `uudo` but may succeed with it, ED/UII automatically invokes `uudo`.

`uudo` inference can also be combined with the implicit-explicit mechanism. When it is the users' intention to open low-integrity files, ED/UII opens the files with low-integrity processes. However, when users do not expect opening the low-integrity files, such openings would be denied. ED/UII considers user-actions such as double-clicking on the files, selecting files from a file-dialog box, or explicitly typing the file names as indications of their intents.

This technique, however, fails if files to be opened depend on the interaction with the programs. Since the files to be accessed are not known at the time of executing program images, ED/UII cannot infer the use of `uudo` in cases such as when high integrity programs are invoked without arguments. A solution is to have the benign programs to capture the user intention to use untrusted files (e.g., when users explicitly selected to open an untrusted file) and spawn a new untrusted process to handle the user request. Handling more general cases, e.g., pipelines, can be addressed using trial execution (Section 6.2.2) or dynamic downgrading (Section 5.5).

Trial-execution based inference

`uudo` inference has the advantage that it is simple— Simply by looking at the parameters used in the `exec` system call, ED/UII can infer what files a program may access. However, there are two problems with this simple technique: The first problem is that command-based techniques do not take into account how files are used. Files specified in arguments may not be necessarily for reading. If the files are for writing only, the inference can wrongly suggest the use of `uudo`. The second problem is that not all files are specified directly in the argument, just as illustrated in the previous example where there are some transformations in the filenames.

Trial-execution based inference relies on the assumption that given the same command, file access behavior of the program would be the same under the same environment. By observing files that are accessed during a trial execution, ED/UII can determine what files will be accessed in the actual execution. ED/UII can then decide what labels to assign to subjects. More importantly, the system can abort directly if the integrity requirement between benign and untrusted principals simply cannot be satisfied. This technique is general enough to support arbitrarily complex command that can involve any number of programs, principals, and files.

Trial-execution relies on programs to exhibit the same file access behaviors whenever the same command is issued. However, actual file accesses can depend on the system environment such as the file systems and environment variables. While ED/UII can execute the command directly on the actual system, this, however, can damage or corrupt system states and leave the system in inconsistent state. Hence, we rely on one-way-isolation to create an isolated environment for "trial" executions. This one-way-isolation environment provides same file system environment, except modifications to the environment are isolated. Changes to the environment are discarded such that the real environment is unaffected. This allows us to capture the actual file access behavior when the same command would have been executed in the system.

During the trial execution, processes and files accessed (in read or write mode) are recorded. These become constraints for the inference process. The goal of the inference is to assign subjects with labels (integrity level) such that all the integrity constraints can be satisfied.

There are various type of constraints based on the interaction policies:

1. *Object principal constraint* corresponds to principals of existing files. Since objects usually do not change their principals, this represents a hard constraint.

2. *Object read access constraint* corresponds to the read accesses captured during the trail-execution. If a process reads from a file of principal F, the principal of the process would need to trust the information from F.

3. *Object write access constraint* are generated from write accesses. This constraint makes sure that processes writing to a file with principal F is allowed to have information flow to F.

The analysis starts by assuming processes can be any principal (benign or untrusted). Based on the constraints, potential process principals will be reduced. For example, if the process read from an untrusted file, then the process must be untrusted to satisfy the read constraint. The analysis completes when all of the information flow constraints are satisfied with some process-to-principal assignment, or the constraints cannot be satisfied. If there exists a principal assignment to processes, the command can then be executed in the actual environment, and appropriate interaction policies will be placed automatically for principal transition. On the other hand, if no solution exists, our system will report to the user that the command cannot be executed.

Note that multiple rounds of trial-execution may be needed to complete the analysis. When an isolated benign process needs to run as untrusted, the system would terminate the execution and restart that process in the corresponding principal. This is because the system cannot let benign processes read untrusted files, even within an isolated environment. Doing so can compromise a benign process, and any read-write access behavior observed can no longer be trusted. Furthermore, the process can also compromise other processes in the isolated environment to affect the inference.

We rely on Linux container for process isolation, `aufs` for creating a copy-on-write file system for isolating processes. Read-write accesses are captured using a library interception framework. We then used XSB [XSB Research Group, 2012] to solve the constraints.

6.3 Related work

Both user-driven access control [Roesner et al., 2012] and our `uudo` inference are based on capturing user intention. However, user-driven access control is about granting resource accesses to untrusted applications. Their focus is on reducing additional user effort for granting these accesses, whereas our goal is to eliminate additional interactions. User-driven access control operates in a hostile environment while `uudo` is for high-integrity processes to infer the use of low-integrity environment. Inaccuracies in their approach can lead to granting authorized permissions to the untrusted applications. In SPIF, inaccurate `uudo` inference will not lead to any security failures.

BLADE [Lu et al., 2010] protects system against drive-by-download malware by stopping unconsented-content executions. It detects user intention of downloading files by parsing file-download dialog-boxes of browsers. Such intent is then used to move files out of the secure zone. Files without user-consent will remain in the secure zone and hence cannot be executed. BLADE relies on user intent for security purposes. Although SPIF also captures user intents, the goals of capturing user intents are different in the two systems. BLADE uses the intent to enforce security policy, but SPIF relies on user intent for improving usability. As such, BLADE requires a kernel module to examine network streams and correlate with files to ensure that file origins are labeled properly. BLADE also needs to monitor keyboard and mouse to make sure that users are actually interacting with the browser rather than a spoofed interface. BLADE has to make sure that user intents are captured properly or otherwise security breaks. On the other hand, SPIF does not rely on user-intent for security, but for usability. SPIF relies on user-intent for `uudo` inference, deciding shadowing policies, as well as the return code for denying opening low-integrity files. Therefore, SPIF does not require kernel driver to make sure the correctness of user-intent.

Chapter 7

Generalized Multi-Provenance Policies

In this chapter, we present a generalization of policies along two dimensions. First, we increase the number of provenance labels supported in our system. Second, polices are no longer constrained to integrity. Principals can also specify confidentiality policies. We describe a language for expressing these richer policies.

The policy language presented in this chapter represents a systematic generalization that can express policies enforced in many previous systems, such as the same-origin policy (SOP) of browsers and the Android policy for apps, as well as many other previous research works [Tiwari et al., 2012].

Our generalized policy is based on ED/UII. We call the policy MultiP.

We start by describing the updated threat model that considers multiple principals (Section 7.1). Then we propose using the notion of permissible information flow to capture both integrity and confidentiality (Section 7.3). After that, we describe a policy language in Section 7.4 that allows each principal to describe its security requirement. Section 7.5, we describe how MultiP can model ED/UII and a simple multi-provenance policy. In Section 7.6, we discuss how principals can interact with each other securely while respecting each principal's security policy. In Section 7.7, we show that MultiP is a general model that can simulate other existing models.

7.1 Threat model in the multiple principal scenario

MultiP builds on the standard multi-user support mechanisms in the OSes to do the provenance tracking and policy enforcement. We assume this mechanism enforces access-control on every resource that belongs to the user, and users are isolated from each other by default, i.e., one compromised user cannot compromise another user.

We assume that system administrators (root/administrator/system) are not malicious. MultiP relies on user-land mechanisms and hence a malicious system administrator can easily bypass our protection.

We assume that whenever new files enter into the system, their network provenance information can be retrieved accurately. For example, when downloading from the Internet, files will be assigned principals that reflect the origin domains if their integrity can be verified (e.g., transferred via HTTPS or checksummed against tampering) by a "trusted" program. Otherwise, we assign the file with an "untrusted" network principal to indicate that the files can be coming from potentially anywhere.

Each principal can define its own integrity and confidentiality policy. For instance, integrity depends highly on applications. A news feed program may trust RSS feeds from a site, but package installers may not consider packages from the same site as high-integrity. By allowing each principal to define its own policy, trust can be context specific. We consider the two most common applications that handle files from multiple sources: web browsers and package installers. We assume that other programs access the network for only information trusted by the program principals. Programs that need to relabel files would need to be aware of MultiP so that they can express to MultiP their trust levels for different files.

We also assume that code from a principal protects the principal itself, i.e., the code has no intention to compromise the security of the code owning principal when executed by the code owning principal. However, code from one principal may act maliciously towards code from another principal. As a result, running code from other principals need to be explicitly allowed in MultiP. Moreover, any principal can be actively trying to compromise other principals. These assumptions allow us to split the principal protection scheme into two parts: cooperative part for protecting the principal itself, and mandatory part for preventing the principal from compromising others.

This is a generalization of the dual-sandbox in the two-principal scenario.

7.2 Supporting multiple principals

ED/UII only supports 2 provenance labels: benign and untrusted. MultiP extends the policy to support multiple principals. MultiP, however, still requires each object/subject to be associated with exactly one principal.

MultiP does not support set notion of principals because permitting joint ownership can lead to an accumulation of principals over time, like the label creep problem, and that it becomes hard to distinguish between consequential and inconsequential contributions to a file. Moreover, when policies of all principals are to be observed, it may not be possible because they may all have different policies. Our goal is to maintain separation between mutually untrusting principals, and it has nothing to do with accurately capturing situations where a file was produced by multiple principals.

7.3 Permissible information flow

The Bell-LaPadula model focuses on confidentiality and enforces no-write-down and no-read-up, with levels concerning secrecy. On the other hand, Biba model focuses on integrity and enforces no-read-down and no-write-up, with levels concerning integrity. It is natural to consider permissible information flow which satisfies both confidentiality and integrity requirement. Specifically, information can flow from principal A to principal B if and only if both of the following conditions are satisfied:

1. Confidentiality of A: A is willing to release information to B. There are three possible cases:
    - A has no secrets, or
    - A trusts B to guard its secrets, or
    - A can declassify the information flowing to B
2. Integrity of B: B is willing to receive information from A. There are three possible cases:
    - B's integrity does not depend on the information, or

- B trusts A that information from A will not compromise its integrity, or
- B trusts itself to sanitize the data from A so as to prevent itself from being compromised

Our two-provenance scenario focuses on integrity— high-integrity principal allows information to flow to low-integrity principal, and low-integrity principal trusts information from high-integrity principal. This creates a unidirectional trust relationship that allows every piece of information to flow from high-integrity to low-integrity.

When we consider secrecy as well, a high-integrity principal may be willing to share all except some confidential information like SSH keys. This constrains the set of information that can flow from high-integrity principals to low-integrity principals, or in general, between any two principals.

We extend ED/UII so that each principal can define its own integrity and confidentiality requirement with respect to other principals using the policy language specified in Section 7.4. MultiP will make sure that any information flow occurs across principals will respect the requirements specified by the principals. Otherwise, the two principals cannot interact.

As we generalize to consider both integrity and confidentiality, we call the benign principal as the `platform` principal, or simply as `platform`.

7.4 Policy language

The basis of security in our approach is isolation: no information is permitted to flow from one principal to another unless both of the principals explicitly allow the flow. Since complete isolation can prevent useful interactions, we provide a policy language for principals to specify their interaction requirements. This language has been guided by the observation that application providers don't want to spend much effort on security policies. Hence we utilize a simple language that uses familiar constructs when feasible (Figure 7.1), together with default policies (Figure 7.3) that work for most of the applications. *It is important to note that the default policies permit a good deal of interaction between principal* `platform` *and other principals.*

File names and network addresses can have wild-cards. Groups of resources can be given a name so that they can be reused in subsequent directives. Some resource groups have predefined meanings:

$$
\begin{aligned}
Exec &= Filename \mid Id && (7.1)\\
Obj &= NetworkEndPoint \mid Filename \mid Id && (7.2)\\
Rule &= (\texttt{allow} \mid \texttt{deny})\ Principal\ (\texttt{read} \mid \texttt{write} \mid \texttt{exec})\ Obj && (7.3)\\
&\mid (\texttt{allow} \mid \texttt{deny})\ (\texttt{read}|\texttt{write}|\texttt{exec})\ Obj\ [\texttt{from}\ Principal] && (7.4)\\
&\mid (\texttt{allow} \mid \texttt{deny})\ \texttt{transition}\ (\texttt{to}|\texttt{from})\ Principal\ [\texttt{via}\ Obj] && (7.5)\\
&\mid Id = \{Obj, ..., Obj\} && (7.6)
\end{aligned}
$$

Figure 7.1: Grammar for generalized policy language

- The group `confidential` is empty by default, but can be defined explicitly. It is used in default policies to prevent other principals from accessing the confidential information.

- The group `config`, `preference`, and `other` correspond to files that are inferred as "config", preference files, and other files based on program access behaviors. Note that the group `config` includes executable files. Principals can define their own policies on whether to share config and preference files using these primitives.

- The keyword `all` includes all principals other than the current principal. When new principals are created in the system, they are automatically added to the group `all`. Principals can also specify specific principals in their policies.

- The keyword `self` refers to the principal itself.

A policy, specified by a principal p, consists of one or more rules that define how p can interact with other principals. Interactions are divided into three categories: read, write, and invoke. MultiP allows each principal to define policies for each of the interaction with respect to other principals.

As discussed previously, a permissible information flow needs to respect both the confidentiality requirement of the information source and the integrity requirement of the information sink. The act of information flow concerns two parties: a subject and an object. Rules are therefore categorized into subject rules (Rule 7.4) and object rules (Rule 7.3). Principal's object rules are applied when that principal's object is involved in information flow. Subject rules are invoked when the principal is a subject. As there are multiple principals, there are also rules governing how principals can transition from one to another (Rule 7.5).

We discuss each of the rules below:

- **Object rules** (Rule 7.3) specify what operations other principals can or cannot perform on the principal's objects. Each of the `allow` or `deny` rule specifies which other principals can read, write, or execute the principal's files. Note that allow/deny rules can refer to file objects as well as network end points. *Obj* must be an object belonging to the principal specifying the policy in order for the rule to be in effect. Read concerns about confidentiality. Write concerns about the integrity of the principal. Execute concerns about the confidentiality of the code.

- **Subject rules** (Rule 7.4) concern operations performed by a subject owned by the principal. Similar to the object rules, there are three interactions that subject rules concern: read, write, and execute. Read and write concern about integrity and confidentiality of the principal. Execute concerns about integrity of the principal when running code from the other principals. These are completely opposite to the object rules.

- **Transition rules** (Rule 7.5) specify if a principal can transition into another principal. Transition rules are useful when the object rules and subject rules between the involving principals cannot be satisfied, and hence may require executing the code as a different principal. There are two types of transition rules: `transition to` and `transition from`. `transition to` specify what principals the principal can transition to. `transition from` specify what principals can transition to the current principal. The rule can also specify an interface *Obj* through which the transition can occur.

An information flow is allowed only if the subject and object rules from both principals have explicitly allowed. These rules are tried in the order in which they are listed. If an access does not match any rule, then it is denied.

1.	`allow`	*untrusted*	`read, exec`	$*$
2.	`allow`	`transition`	`to`	*untrusted*

(a) Policy for benign

3.	`allow`	`read, write, exec`	$*$	`from`	*benign*
4.	`allow`	*benign*	`read, write, exec`	$*$	
5.	`allow`	`transition`	`from`	*benign*	

(b) Policy for untrusted

Figure 7.2: Policy for two-provenance case

7.5 Example Policies

7.5.1 ED/UII

Figure 7.2 captures the ED/UII policy descried in previous chapters. Rule 1 allows information to flow out of the benign principal without any confidentiality concerns. Rule 3 allows untrusted to read from benign without integrity concerns. Rule 1 and 3 combined allow information to flow from the benign to the untrusted principal. Rule 2 and 5 are transition rules. They allow benign to transition into untrusted.

Note that Rule 4 does allow benign principal to read its files. However, MultiP would not consider this as a permissible information flow because there is no corresponding rule in the benign principal allowing it to read from untrusted. Permissible information flow requires both the involving principals willing to permit the flow.

7.5.2 A simple multi-provenance policy

We now extend the two-provenance policy from Figure 7.2 to support multiple, mutually untrusting principals. Recall that in this case, we use the *platform* principal in the place of *benign* principal.

Figure 7.3 shows the policy for the `platform` principal. This policy is the same as the policy for the benign principal (Figure 7.2a) except (1) with the confidentiality rules (Rule 1 and 2) added, and (2) allowing transition

1.	confidential = {.ssh/∗, ...}			
2.	deny	all	read	confidential
3.	allow	all	read, exec	∗
4.	allow	transition	to	any

(a) Platform principal's policy

5.	confidential = {}				
6.	deny	all	read	confidential	
7.	allow	all	read, exec	∗	
8.	allow	transition	to, from	any	
9.	allow	read, write, exec	∗	from	*platform*

(b) Default policy for other principals

Figure 7.3: Policy for multi-provenance case

to any principal rather than just untrusted. This is because the platform principal forms the basis for other principals to interact. Therefore, code, configuration files, preference files and data from the platform are readable to other principals under the default policy.

A remote principal p, in the simplest case, does not provide a policy, and thereby ends up using default policy shown in Figure 8.9d. Note that this policy does not provide any security from platform code (Rule 9). This is intentional: since platform code runs with higher privileges (possibly root), it is not always possible to protect p from the platform. The default policy also allows other principals to read and execute all except confidential files of p (Rule 5, 6, and 7). By defining the list confidential, p can very easily control the privacy of its data. Rule 8 allows transitioning cross any remote principals. Rule 4 allows benign principal to transition to any remote principal.

7.6 Interaction between principals

While permissible information flow focuses only on flows between information sources and information sinks, a program execution would in general consist of multiple information flows involving several principals. There can be four entities involved in a program execution:

- Invoker principal I: the principal that wants to execute a piece of code
- Code principal C: the principal that owns the code
- Read data principals D_R: the set of all principals that own the data read during the execution
- Write data principals D_W: the set of all principals that own the data written during the execution

Since each principal defines its own policy with respect to other principals, MultiP will select an executing principal P to run the code such that all of the rules defined by the principals are observed. We use the notation $A \rightarrow B$ to denote that an information flow from principal A to principal B is permitted by both their policies.

Let I denote a principal invoking the execution of code owned by a principal C. Let D_R and D_W be the principals that own the files read and written during this execution. Then, our system will attempt to find a principal P such that the following flows are permitted, and execute C within a process owned by P:

- $I \rightarrow P$ (Invoker)
- $C \rightarrow P$ (Code owner)
- $\forall D_r \in D_R \;\; D_r \rightarrow P$ (Read data owner)
- $\forall D_w \in D_W \;\; P \rightarrow D_w$ (Write data owner)

Consider a typical desktop OS scenario with two principals: *user* and *root*, with *root* $\rightarrow$ *user* for files such as code. Suppose that the user invokes a root owned program that modifies a user file. MultiP considers *user* as the invoker principal I that triggers the action. The code is owned by root, i.e., $C =$ *root*. When the code runs, it will read both root-owned files (e.g., libraries) and user-owned files (e.g., preference files), so $D_R = \{user, root\}$. The execution can result in modifying user files, i.e., $D_W = \{user\}$. As a result, the executing principal can only be *user*.

Satisfying the first three constraints are necessary for a program to execute. This is because the act of invoking the code and reading code and data characterizes a program execution. Permissions to write to file is not essential, as the modification can be shadowed. (However, as discussed before, some user input or policy inference is necessary to determine whether shadowing is appropriate, or if the whole operation should be denied.)

7.6.1 Principal resolution algorithm

MultiP allows only permissible information flows. Flows that violate principals' policy will be denied. To minimize the impact on usability, MultiP attempts to choose to run code with a principal that satisfies all the information flow requirements.

There are 4 possible choices for the executing principal P. It can be the same as I, C, D_r/D_w, or other. We summarize the implication of each choice below:

1. **Desktop mode when the execution runs as principal Invoker I.**
   This corresponds to the conventional usage of desktop application, where code simply runs with the privilege of the invoker, regardless of who owns the code.

2. **App mode when the execution runs as the code principal C.**
   App model on mobile OSes such as Android, where each app is runs with its own privilege (similar to setuid on desktop). Any app on Android can invoke any other app to perform any action.

3. **Data-oriented mode when the execution runs as the data owning principal D_r/D_w.**
   Bubbles [Tiwari et al., 2012]-like system which organizes the system based on data labels. Each data is tagged with a label. Code and invokers are simply tools for processing data. The same app can produce data with different labels.

4. **Hosted mode when the execution runs as a principal none of the above.**
   This is similar to the ephemeral container in Apiary [Potter and Nieh, 2010], LXC, or Alcatraz [Liang et al., 2009], where the code runs in isolation. MultiP runs the code with a principal other than I, C, or D_r/D_w. The process and its output would not be accessible by other principals.

The goal of the different interaction modes is to allow principals to interact while respecting their interaction policy. The algorithm to decide on which mode to run is simply test if each of the mode listed above observes the policies of the involving principals. The testing order is as follow: Desktop, App, Data-oriented, and finally Hosted. The order is determined by the impact on usability for legacy desktop applications.

An alternative view is that principals in MultiP form a partially ordered set, with a binary relationship defining permissible information flow between two principals. In the simplest scenario, we consider principals higher in the set allow information to flow to principals lower in the set. The first three constraints imply that P is at most the greatest lower bound among $\{I$, C, and $D_r, \forall D_r \in D_R\}$. Such P may or may not exist, depending on the permissible information flow between the principals. If no such P exists, the execution will be denied as the information flow requirement between principals cannot be satisfied. Otherwise, the execution will be allowed to run as principal P.

When shadowing data is not an option, the forth constraint translates into $P \geq D_w$, $\forall D_w \in D_W$. If no such P exists, the execution will be denied.

7.7 Simulating existing models

7.7.1 Bubbles

Bubbles [Tiwari et al., 2012] isolates resources based on *context*, which is an abstraction to capture events based on contacts and time. For example, a context can be a conference event with a group of participants at specific time. Before using an app, users need to select the context that they want to use. Resources created in a context belongs to the context and can only be accessed by that context.

Both Bubbles and our system support isolation. In Bubbles, apps are tools for manipulating resources, but they do not own data. MultiP can simulate Bubbles by mapping a principal into every Bubbles' context. Since Bubbles does not allow contexts to interact, this corresponds to no interaction between principals in MultiP.

Bubbles require extra efforts to develop trusted viewers to let users to browse and select contexts. These viewers are trusted not to steal information from different contexts and not to corrupt resources within the contexts. They assemble information from different contexts and present users an unified view. For example, images can belong to different bubbles and each has a different provenance label. A trusted viewer is needed to view images from all bubbles.

While simulating the Bubbles's model, MultiP also need to assemble information from different contexts. Unlike Bubbles, MultiP can reuse existing desktop applications to act as trusted viewers instead of developing new applications. Indeed, simply the file manager (e.g., Windows Explorer) itself running as the *platform* principal can already browse all the files created in each context. MultiP transparently merges files created from different principals together. By double-clicking on a file, the principal corresponding to the context will start and let the user to view the context.

```
1.   allow   read         other   from   all
2.   allow   transition   to      all
```

(a) Policy for trusted viewer

```
3.   allow   trusted_viewer   read   other
4.   allow   transition       from   trusted_viewer
```

(b) Policy for contexts

Figure 7.4: Policy for modeling Bubbles
with principal P using coding from R

Figure 7.4 shows a MultiP policy for simulating Bubbles. The policy consists of two parts: one for trusted viewers and one for contexts. The trusted viewers are trusted to view data from all principals so that they can present users a summarized view. The policy allows the trusted viewer to read using Rule 1 and 3. The transition rules (Rule 2 and 4) allow the trusted viewers to transition into different context when users want to dive in. The policies for the contexts simply permit the trusted viewers the accesses.

7.7.2 Android

Android isolates applications based on app origin. App origin is identified by the developer's key used to sign the apps. Apps from the same origin run as the same user and are free to interact using existing Linux IPC mechanisms such as signaling to interact. Apps from different origins run with different userid, and they can only interact with other apps using Android's own sharing mechanisms such as *Intent*.

Our system can simulate Android by mapping each app origin as a different principal. In Android, apps can decide what data it want to share and receive. Hence, private files (such as code, preference, and configuration files) are not shared in Android. In our system, all data files (`other`) can be shared with other principals unconditionally, while code, configuration, and preference files are not visible by other principals and hence are private.

On Android, user can select a particular application which she wants the data to be shared with. The user can also select a particular app if multiple apps can handle the intent during the *intent resolution phase*. On our system, the Windows Explorer serves as the explicit sharing mechanism across principals. Users can use *open as* to select a desired app to consume the data. Applications that can accept the data would have handlers registered with the Windows Explorer/Shell.

```
1.    allow     transition  to,from      all
```

Figure 7.5: Policy for modeling Android app model

Figure 7.5 shows the policy in MultiP for modeling Android app. Every app in Android uses the same policy: it allows transitioning into or from other apps (rule 1). This is because apps can freely create an intent to invoke other apps. Intent filter in the Android model can filter what apps to invoke based on data type, and is captured by specifying what objects they want to share. By default, every data file is accessible using `other`. Apps can also define their own data type that they want to share.

The MultiP policy models the Android app model. However, when a piece of data from one app is malicious, the malicious data can spread to another app and compromise that app. This is a problem in Android. To solve this problem, one can allow an app to execute the code of another app in an one-way isolation environment. The policy in Figure 7.6 allows this one-way isolation. Note that Rule 1 and 2 permit other apps to read any file, including config, preference, and data files. Transition rules (Rule 3 and 4) are replaced by allowing executing executables. This creates an isolation environment for the app to run code from another app: Data from a malicious app cannot spread to other apps.

1.	`allow`	`all`	`read`	$*$
2.	`allow`	`read`	$*$	
3.	`allow`	`all`	`exec`	`executables`
4.	`allow`	`exec`	`executables`	

Figure 7.6: Policy for modeling Android app model for each app principal

7.7.3 Web

The Web security model also adopts isolation. It applies same-origin-policy (SOP) to isolate based on origins, defined by the tuple ¡domain name, protocol¿. Resources for each principal in the SOP model includes code (JavaScript), DOM and local storage objects, and remote resources (accessed via cookies).

There a few modes of interactions in the web model. We discuss each of them and how they can be modeled by MultiP:

- **SOP** does not allow a principal P to access resources belonging to other principals (e.g., R). P can include the entire page from R but cannot access DOM or local storage objects from R. Cookies from R are also not accessible by P. However, SOP allows resources such as JavaScript and images from R to be loaded by P. P can use JavaScript and images from R as if they are from P. To simplify our discussion, we assume that JavaScript can be loaded from R.

  Principal P can include code R as libraries using `script` tags. The code will then be executed as if it belongs to P, i.e., it will have all the privileges that P has. The code, however, cannot access local or remote resources from R since P does not have accesses.

Modeling SOP in MultiP is simply allowing principal P to execute code from principal R. This is done by using the policies in Figure 7.7. They keyword `JavaScript` consists of a the set of code that R is willing to share with P. Note that SOP prevents a principal from reading code of another principal.

```
1.    allow      all         exec          JavaScript
2.    allow      exec        JavaScript    from   all
```

Figure 7.7: Policy for modeling SOP policy enforced by browsers

With the policy in Figure 7.7, MultiP would allow P to execute code from any other domain. P can execute the code and use the code from R as library. However, there is no domain transition rule from P to R. P cannot access data in R too.

When P executes code from R, the trust assumption is that P trusts the code from R, and R can access every resource in P.

- **CORS (Cross-Origin Resource Sharing)** is a mechanism for relaxing the limitation of SOP. SOP limits code from principal P to access only P's resources. It is something useful for P to access other's resources as well, such as when R is a content provider for application at P. Since the SOP policy is enforced by web browsers to protect principals, the relaxation is actually done by the browsers. When code from P requests resources from R (e.g., using Ajax), the browsers will consult R first using pre-flight requests to check if R is willing to give P access to its resources. Only if R allows P to access the resources, the request from P can reach R. Unlike SOP which only allows script tags (basically only HTTP GET requests), CORS allows also POST and PUT requests that can modify resources on R. CORS has recently been standardized.

  CORS protocol itself is coarse-grained that it either allow or deny P to make requests to P. R can implement additional logics to decide whether to serve each of the request made from P. Hence, the MultiP

1.	allow	all	exec	JavaScript	
2.	allow	exec	JavaScript	from	all
3.	allow	read, write, exec	$*$	from	permitted_domains
4.	allow	permitted_domains	read, write, exec	$*$	

Figure 7.8: Policy for modeling CORS
policy enforced at the browser, with permitted_domains retrieved from remote servers

policy specified in Figure 7.8 would allow P to access any files of R. Rule 1 and 2 are from SOP, which allows executing JavaScript from any principal. Rule 3 and 4 uses a special keyword permitted_domain to capture the domains that are willing to grant P access to its resources. The set of permitted_domain is managed by browsers using pre-flight requests in CORS. Browsers enforce the policy in Figure 7.8. The web server of R can make further decisions as to whether to serve the requests. This is not enforced by browser and hence not captured by the policy in Figure 7.8.

1.	allow	all	exec	JavaScript	
2.	allow	exec	JavaScript	from	all
3.	allow	transition	to	permitted_domains	
4.	allow	read	$*$	from	permitted_domain

(a) Policy for P, enforced by browsers

5.	allow	transition	from	P	via predefined_interface
6.	allow	P	read	predefined_interface	

(b) Policy for R

Figure 7.9: Policy for modeling CORS interacting with specific interfaces with principal P using resources from R

To model the interaction between browsers and the web server of R, MultiP can model the decision logic in R by permitted P to access resources of R via a single mediation point (e.g., predefined_interface), which can then control whether to serve the requests. This can be achieved using the policy in Figure 7.9. The policy is divided into two parts: Figure 7.9a is enforced by the browser on P and Figure 7.9b is enforced by the web server R. Rule 1 and 2 are from the SOP. Rule 3 transits the control to the web server owned by principal R, combining with Rule 5, P can transition to R by invoking specific interfaces specified in `predefined_interface`. Rule 4 and 6 allow P to read the result back from the web server of R. R can perform mediations at `predefined_interface` as it is the single entry point for P to access resources in R.

- **JSONP** requires no modification to the HTTP protocol in order to realize cross origin sharing on top of SOP. JSONP exploits the fact that SOP allows code from other principals to be accessible via `script` tag, i.e., Rule 1 and 2 in Figure 7.7. By having the other principal R to return data resources inside a JavaScript file, P can access the data resources prepared inside the JavaScript from R. JSONP has been used in frameworks such as jQuery, where the code running as principal P can specify a callback method name in the request to R. The script returned from R can then invoke the callback method with the data requested.

  JSONP is considered as exploiting a loophole in SOP. The script returned from R could access all resources of P, including P's DOM tree, local storage, and remote resources of P as cookies are accessible too. This is because P runs the code from R using P's privilege. If the principal R is malicious or compromised, SOP can provide no protection to P.

  To provide better protection, a browser can create an ephemeral principal and runs the script returned from R to construct the data object. After the data object is constructed, P can then retrieve the data object from the ephemeral principal. Figure 7.10 captures this policy. The policy is enforced by the browser and is divided into two parts: Figure 7.10a is a policy for principal P. Figure 7.10b is a policy for an ephemeral principal (P, R) instantiated when P would like to access resources from R using JSONP. Rule 1 and 2 are from the SOP. Rule 3 allows P to transition into the ephemeral principal (P, R). Rule 5 allows the ephemeral principal to run code from R. Rule 4 and 6 allow P to obtain the result from the code. This is a much safer option for realizing JSONP because if R is malicious, the principal (P, R) can only access the code from R, but not resource from P.

```
1. allow  all         exec        JavaScript
2. allow  exec        JavaScript  from    all
3. allow  transition  to          ephemeral_domain(P,R)
4. allow  read        other       from    ephemeral_domain(P,R)
```

(a) Policy for domain P accessing data from R

```
5.   allow   exec   JavaScript   from    R
6.   allow   P      read         other
```

(b) Policy for `ephemeral_domain(P,R)`

Figure 7.10: Policy for modeling JSONP with isolation policy enforced at the browser

- **URL.hash** [Wang Jiaye, 2011] is another method for allowing principals from two origins in same window but different frame to communicate despite of SOP. Browsers allows code to access the URL location of another frame. Two principals can be loaded into two different frame and communicate with each another by changing the hash tag value at the end of their URL. Since hash tag in the URL does not trigger any page reload, it can be served as a shared object for principals to communicate. A participating principal would monitor hash tag values of the other principals, and then modifies its own hash tag value in respond.

  Figure 7.11 captures the policy that browsers enforce to realize URL.hash communication. Note that browsers enforces additional policy rules (Rule 3 and 4) over the SOP policy to allow resources, i.e., `window.location`, to be readable by other domain. Since this mechanism is not designed for inter-principal communication in the first place, there is no security protection. Any other principals can access the hash tag values and intercept the communication.

```
1.  allow   all    exec              JavaScript
2.  allow   exec   JavaScript        from              all
3.  allow   all    read              window. ∗ .location
4.  allow   read   window. ∗ .location
```

Figure 7.11: Policy for modeling URL.hash
policy enforced at the browser

The mechanism provides no way for principals to protect their communications from getting sniffed. MultiP can easily protect the `window.location` by limiting the accesses to the intended principals (Changing Rule 3 and 4 to principal specific). Figure 7.12 shows the URL.hash policy that allows R to observe URL.hash of P. The browser ensures that only intended principals, i.e., R, can access P's location information.

```
1.  allow  all    exec                JavaScript
2.  allow  exec   JavaScript          from                all
3.  allow  R      read                P.window.location
4.  allow  read   R.window.location
```

Figure 7.12: Policy for modeling URL.hash with protection
policy enforced at the browser for P, permitting P and R access URL.hash

- **Post-message** is a newer standard for web principals to interact. Its idea is similar to the URL.hash method. Post-message allows principals to communicate by sending messages. When sending a message, the sending principal can specify the receiving principal using origin. A callback function provided by the receiving principal will be invoked. The callback function can also check the message originating principal. This push bashed mechanism is much more secure than the URL.hash method.

  Figure 7.13 captures the post-message model. It designate R.`window.message` as the object for P to communicate with R. Post-message is similar to URL.hash except post-message allows other principals to send messages to the shared resources, while URL.hash allows other principals to read messages from the shared resources. Hence, the differences be-

```
1.  allow    all     exec              JavaScript
2.  allow    exec    JavaScript        from               all
3.  allow    all     write             P.window.message
4.  allow    write   *.window.message
```

Figure 7.13: Policy for modeling post-message policy enforced by browsers, with principal P sending messages

tween the URL.hash policy (Figure 7.13) and the post-message policy are (1) the operation changed from `read` to `write`, and (2) the resources are `window.message`.

In post-message, principals can check the origin of the messages before deciding how to handle the messages. We can use similar policy in CORS with specific interface (Figure 7.9) to transition into another principal and only allow the principals to communicate via specific interfaces.

- **MashupOS Sandbox abstraction** [Wang et al., 2007] is one of the abstraction that has not been implemented in browsers. The <Sandbox> abstraction allows one principal *Integrator* to enclose another principal *Provider*. The *Integrator* can access the content and code of *Provider* without worrying getting compromised by *Provider*. Instead of running the enclosed content as *Integrator* or *Provider*, it runs the content as `unauthorized`. This protects both the *Integrator* and the *Provider*.

  Figure 7.14 shows the policy for modeling the MashupOS Sandbox abstraction. The modeling involves three principals: *Integrator*, *Provider*, and `unauthorized`. `unauthorized` is an ephemeral principal. Rules 1, 2, 4, and 5 are SOP. A feature of the sandbox abstraction is to allow the integrated code to run securely by running as the principal `unauthorized`. MultiP models this by allowing transitioning between *Integrator* and `unauthorized` via specific interfaces `DOM_method_calls` using Rules 3 and 8. The `unauthorized` principal has access to code and resources of the enclosing page (Rules 6 and 7), as well as running code from the *Integrator* (Rule 9) so that the results can be presented.

1.	`allow`	`all`	`exec`	`JavaScript`	
2.	`allow`	`exec`	`JavaScript`	`from`	`all`
3.	`allow`	`transition`	`to, from`	`unauthorized`	

(a) Policy for *Integrator*

4.	`allow`	`all`	`exec`	`JavaScript`	
5.	`allow`	`exec`	`JavaScript`	`from`	`all`

(b) Policy for *Provider*

6.	`allow`	`exec`	`JavaScript`	`from`	*Provider*
7.	`allow`	`read`	`other`	`from`	*Provider*
8.	`allow`	`transition`	`to, from`	*Integrator*	`via DOM_method_calls`
9.	`allow`	`exec`	`JavaScript`	`from`	*Integrator*

(c) Policy for `unauthorized`

Figure 7.14: Policy for modeling MashupOS Sandbox abstraction policy enforced at the browser, *Integrator* sandboxes *provider*

Part III

Systems

Chapter 8

Spif

In this chapter, we present Spif, a malware defense system based on provenance tracking.

The scale and sophistication of malware continues to grow exponentially. The reactive approach embodied in malware scanners and security patches is no match for today's stealthy, targeted attacks. Recognizing this fact, researchers as well as software vendors have been developing *proactive techniques* that can protect against previously unseen exploits and/or malware attacks. These techniques can be classified into three main categories: *sandboxing, privilege separation,* and *information flow control.*

Sandboxing techniques [Goldberg et al., 1996, Provos, 2003, Ubuntu, 2015, Yee et al., 2009] mediate all security-relevant operations performed by applications, permitting only those deemed "safe" by a sandboxing policy. The scope of damage that can result from a malicious (or compromised) application is hence limited by this policy. On UNIX, applications frequently targeted by attacks are typically protected using SELinux [Loscocco and Smalley, 2001b] or AppArmor policies [Ubuntu, 2015]. Microsoft Office used a sandbox for its *protected view* [Microsoft, 2015b]. This sandbox ensures that a compromised process cannot overwrite system or user files or registry entries.

Privilege separation techniques [Provos et al., 2003] refine the sandboxing approach to support applications requiring significant access to realize their functionality. The application is decomposed into a small, trustworthy component that retains significant access, and a second larger (and less-trusted) component whose access is limited to that of communicating with the first component in order to request security-sensitive operations. While sandboxes can confine malicious as well as frequently targeted benign applications (e.g., browsers), privilege separation is applied only to the latter class. Chromium browser [Reis and Gribble, 2009], Acrobat Reader and Internet Explorer are some of the prominent applications that employ privilege separation, more popularly known as the *broker architecture.* These applications sandbox their renderers, which are complex and are exposed to untrusted content. As a result, vulnerabilities in the renderer (or more generally, a *worker*) process won't allow an attacker to obtain all privileges of the user running the application.

Information flow control (IFC) techniques [Biba, 1977, Zeldovich et al., 2006, Krohn et al., 2007, Li et al., 2007, Sun et al., 2008b, Mao et al., 2011, Sze et al., 2014] maintain labels on files and processes to keep track of the flow of sensitive and/or untrusted information in the system. Classical integrity policies such as the Biba policy [Biba, 1977] enforce both no-read-down (i.e., integrity-critical applications cannot read untrusted data) and no-write-up (i.e., untrusted applications cannot create or overwrite high-integrity files) policies. In contrast, *Windows Integrity Mechanism* (WIM) [Microsoft, 2015c] enforces just the no-write-up policy. Indeed, WIM is primarily deployed as a sandboxing mechanism: progressively more restrictive policies are enforced on lower integrity processes, while high-integrity processes are unconfined. In contrast, the strength of integrity protection in IFC stems from policy enforcement on high-integrity processes, which prevents them from compromised by consuming untrusted data or code.

8.1 Challenges

Application of these three approaches for malware defense poses several technical as well as practical challenges.

Policy development Policy affects both usability and functionality of applications. Restrictive policies can block more attacks, but they also tend to break applications. Moreover, policy development requires not only a good understanding of applications, but also the OS semantics. A recent Adobe Reader XI vulnerability [Google Security Research, 2014] exploits the semantics of *junctions* on NTFS, where the broker process failed to sanitize paths and ended up allowing workers to create files at arbitrary locations.

Application and OS compatibility To run successfully with a policy and its enforcement framework, applications need to be re-architected, or at a minimum, be made aware of the confined environment. Most IFC approaches require nontrivial changes to applications as well as the OS. There have been efforts to automate some of the steps (e.g., automating privilege separation [Brumley and Song, 2004]) or to minimize application changes for IFC (e.g., PPI [Sun et al., 2008b]), but in practice, most techniques end up requiring substantial effort in rewriting or porting applications or the OS.

Sandbox escape attacks Given the large effort needed to (a) develop policies and (b) modify applications to preserve compatibility, it is no wonder that in practice, confinement techniques are narrowly targeted at a small set of highly exposed applications. This naturally leads attackers to target sandbox escape attacks: if the attacker can deposit a file containing malicious code somewhere on the system, and trick the user into running this file, then this code is likely to execute without confinement (because confinement is being applied to a small, predefined set of applications). Alternatively, the attacker may deposit a malicious data file, and lure the user to open it with a benign application that isn't sandboxed. In either case, the attacker is in control of an unconfined process that is free to carry out its malicious acts.

As a result of these factors, existing defenses only shut out the obvious avenues, while leaving the door open for attacks based on evasion (e.g., Stuxnet [Falliere et al., 2011]), policy/enforcement vulnerabilities (e.g., sandbox escape attacks on Adobe Reader [Fisher, 2014], IE [Li, 2015] and Chrome [Constantin, 2013]), or social engineering. Stuxnet [Falliere et al., 2011] is a prime example here: one of its attacks lures users to plug in a malicious USB drive into their computers. The drive then exploits a link vulnerability in Windows Explorer, which causes it to resolve a crafted *lnk* file to load and execute attacker-controlled code in a DLL.

8.2 Approach overview and key features

We present a new approach and system called SPIF, which stands for Secure Provenance-based Integrity Fortification, to achieve OS-wide integrity protection on contemporary OSes. Unlike previous approaches, SPIF:

- *Requires no manual effort for policy development.*

- *Requires no application or OS modifications,* being able to support Linux, BSD, and all major versions of Windows since Windows XP, and feature-rich, unmodified applications such as MS Office, IE, Chrome, Firefox, Skype, Photoshop, and VLC.

- *Confines all applications,* thereby taking away the motivation for sandbox escape attacks.

Spif defends against unknown malware attacks targeting integrity[1], including stealthy malware such as Stuxnet and Sandworm [Ward, 2014].

Spif uses provenance tracking to track integrity. We define integrity based on the origin ("where") of a piece of information. Our implementation classifies origins into just two categories: *benign* and *untrusted*.

Spif applies the dual sandboxing architecture (Chapter 4) to enforce the ED/UII policy (Chapter 6). Spif therefore:

- tracks provenance reliably without modifying OSes or applications (Chapter 4)

- enforces policies robustly (Chapter 4)

- preserves user experience (Chapter 6)

- automates policy development (Chapter 6)

8.3 Threat model

We state the threat model for Spif. It is mostly the same as in Section 2.2.

We assume that users of the system are benign. Any benign application invoked by a user will therefore be non-malicious. If a user is untrusted, Spif can simply treat the user as an untrusted user and every subject created by that user is of low-integrity.

Spif assumes that any files received from unknown or untrusted sources will be labeled as low-integrity. This can be achieved by exclusion: Only files from trusted sources like OS distributors, trustworthy developers and vendors are labeled as high-integrity. All files from unverifiable origins (including

[1]Although Spif does not focus on confidentiality, note that most malware needs to embed itself into the system in such a manner that it would be invoked automatically. This step requires compromising system integrity, and will be caught by Spif.

network and external drives) are labeled as untrusted. As described later, Spif's labeling of incoming files has been seamlessly coupled with Windows Security Zones, which has been adopted by all recent browsers and email clients. For Unix systems, we developed browser and email client add-ons to label files. An administrator or a privileged process can upgrade these labels, e.g., after a signature or cryptographic hash verification. We may also permit a benign process to downgrade labels.

Spif focuses on defending attacks that compromise the system-integrity, i.e., performing unauthorized modifications to the system (such as malware installing itself for auto-starting) or environment that enables the malware to subvert other applications or the OS (e.g., `.bashrc`). Although Spif can be configured to protect confidentiality of user files, this requires confidentiality policies to be explicitly specified, and hence we did not explore it further in this paper. It should be noted that files containing secrets useful to gain privileges are already protected from reads by normal users. This policy could be further tightened for untrusted subjects.

We assume that benign programs rely on system libraries (i.e., `libc.so`, `ntdll.dll` or `kernel32.dll`) to invoke system calls. Spif intercepts system calls from the libraries to prevent high-integrity processes from accidentally consuming low-integrity objects. We *do not* make any such assumptions about untrusted code or low-integrity processes, but do assume that OS permission mechanisms are secure. Thus, attacks on the OS kernel are out of the scope of Spif.

8.4 Design

Spif leverages the dual-sandboxing architecture in Chapter 4 as its enforcement mechanism to confine both benign (Section 4.3) and untrusted processes (Section 4.2). Spif enforces ED/UII policy (Section 5.2) and supports policy inference (Section 6) to preserve normal desktop user experience. Spif can provide the integrity and availability guarantees stated in Section 5.6.

8.5 Implementation

In this section, we discuss how we implemented Spif on Ubuntu, PCBSD, and Windows. Our primary implementation was performed mainly on Ubuntu

10.04 and Windows 8. We ported Spif to PCBSD, one of the best known desktop versions of BSD, to illustrate its feasibility on BSD system. In addition, we also tested our system on Windows XP, 7, and Windows 10.

Spif requires the initialization in Section 4.5 for initial file labeling and realization of the dual-sandbox architecture.

We discuss below how Spif integrates with OSes in terms of initial file labeling, relabeling, display server/DBus, and file utilities.

8.5.1 Initial file labeling

An important requirement for enforcing policies is to label new files according to their integrity. Some files may arrive via means such as external storage media. In such a case, we expect the files to be labeled as untrusted (unless the authenticity and/or integrity of files could be verified using signatures or other means). However, we have not implemented any automated mechanisms to ensure this, given that almost all files arrive via the Internet.

Web browsers We designated `Firefox`, the main web browser on Ubuntu, to protect itself from network inputs and inputs from local files selected using a file dialog by the user. Files selected by user using a file dialog are mainly used for uploading. These files are identified by the "implicit-explicit" mechanism described in Section 6.1, preventing `Firefox` from using untrusted files as non-data inputs. To ensure that downloaded files are associated with the right integrity labels, we have developed a `Firefox` add-on, which uses a database to map domains to integrity levels.

As a second alternative, we dedicated an instance of the web browser for benign sites. Using policies, the benign instance can be restricted from accessing untrusted sites. In Spif, we manually defined a white-list of benign sites. A better alternative would use white-lists provided by third parties. Instead of blocking users from visiting untrusted sites, we can invoke the untrusted browser instance to load the pages directly.

Email clients Email clients introduce untrusted data into the system through message headers, content, and attachments. Our approach is to trust the email reader to protect itself from untrusted sources. However, attachments are given labels corresponding to the site from which the attachment was received. We have developed an add-on for `Thunderbird` on Ubuntu for this purpose. However, the current email protocol (SMTP) does not protect against spoofing. To provide trustworthy labeling, we could either rely on digital signatures (when present), or on the chain of SMTP servers that handled the email. Such spoof-protection has not yet been implemented.

Integrating with Windows Security Zone SPIF's integration with Windows leverages a Windows built-in mechanism called Windows Security Zones. Instead of developing add-ons, most web browsers and email clients such as Internet Explorer, Chrome, Firefox, MS Outlook, and Thunderbird automatically assign security zones when downloading files. It is a piece of information stored in Alternate Data Stream, along with the file. The origins-to-security zones mapping can be customized. Windows provides a convenient user-interface for users to configure what domains belong to what security zones. Microsoft also provides additional tools for enterprises to manage this configuration across multiple machines with ease.

Windows has used security zone to track origin, but in an ad-hoc manner. When users run an executable that comes from the Internet, they are prompted to confirm that they really intend to run the executable. Unfortunately, users tire of these prompts, and tend to grant permission without any careful consideration. While some applications such as Office make use of the zone labels to run themselves in protected view, other applications ignore these labels and hence may be compromised by malicious input files. Finally, zone labels can be changed by applications, providing another way for malware to sneak in without being noticed.

SPIF makes the use of security zone information mandatory. SPIF considers files from `URLZONE_INTERNET` and `URLZONE_UNTRUSTED` as low-integrity. Applications *must* run as low-integrity in order to consume these files. Moreover, since SPIF's integrity labels on files cannot be modified, attacks similar to those removing file zone labels are not possible.

Software Installation Our system relies on correct integrity labeling when new files are introduced into the system. Of particular concern is the software installation phase, especially because this phase often requires administrative privileges. Solutions have previously been developed for securing software installation, such as SSI [Sun et al., 2008a]. We are implementing an approach similar to SSI to protect the software installation phase and to label files introduced during the installation on Ubuntu. SPIF can then enforce the policies at run time based on the labels.

Rather than safeguarding the installation process, approaches have been developed to eliminate the installation phase completely. 0install [Leonard et al., 2015] allows users to execute a software directly using a url. Application files are cached entirely in user's home directory. We tested our system with 0install. It allows users to directly execute a remote application securely simply based on a url. 0install supports multiple platforms, including Linux and Windows.

8.5.2 Relabeling

Spif automatically labels files downloaded from the Internet based on its origin. However, it is possible that high-integrity files are simply hosted on untrusted servers. As long as their integrity can be verified (e.g., using checksum), Spif would allow users to relabel a low-integrity file as high-integrity. Changing file integrity level requires copying the file from redirect storage to the main file system, while the file ownership is changed from R_U to R. We rely on a trusted application for this purpose, and this program is exempted from the information flow policy. Of course, such an application can be abused: (a) low-integrity programs may attempt to use it, or (b) users may be persuaded, through social engineering, to use this application to modify the label on malware. The first avenue is blocked because low-integrity applications are not permitted to execute this program. The second avenue can be blocked by setting mandatory policies based on file content, e.g., upgrading files only after signature or checksum verification.

8.5.3 Display server and DBus

Resources such as display (X-Server or desktop window) and DBus need to be shared by both benign and untrusted processes for usability, yet these mechanisms support very little or no access control once processes are granted access the resources. An untrusted GUI application can send arbitrary events (key events or Windows events) to another benign GUI programs. An untrusted process can also send messages to other programs listening to the user DBus.

Spif attempts to protect these resources using two methods: isolation or enforce policies to restrict operations. In isolation, Spif creates a redirected copy of the resource and let untrusted processes connect to the redirected copy. On Ubuntu, Spif uses `Xephyr`, a nested X-server to serve untrusted processes. As for DBus, Spif transparently redirects untrusted processes to connect to an untrusted DBus server.

Another alternative is to enforce policies to restrict interactions between untrusted processes and the server. Spif uses X-security-extensions to designate untrusted processes as *untrusted X-client*, to restrict/disable accesses to certain X resources. DBus does not provide built-in mechanisms to designate clients as untrusted. We have also built a DBus proxy which intercepts DBus messages between server and untrusted processes. This allows Spif to enforce policies on DBus. Since this option trusts the X-server or the DBus proxy, it is not as secure as the first alternative, but integrates smoothly in terms of user experience.

Our implementation do not consider Windows messages because any process with a handle to the desktop can send message to any other process on the desktop, regardless of the userid of the processes. This is demonstrated in shatter attack. As a result, an untrusted process can send Windows messages to a benign process. Windows servers support multiple concurrent users, Spif could use remote-desktop to achieve isolation similar to `Xephyr`. For policy-based enforcement, there are two techniques to solve the problem: The first technique is to apply job control in Windows to prevent untrusted processes from accessing handles of benign processes. By setting the `JOB_OBJECT_ULIMIT_HANDLES` restriction, a process cannot access handles outside of the job. The other method is to run untrusted processes as low WIM integrity processes. WIM already prevents lower integrity processes from sending messages to higher integrity processes.

8.5.4 File utilities

Files belonging to different integrity levels co-exist. Utilities such as `mv`, `cp`, `tar`, `find`, `grep`, and `rm` may need to handle files of high and low integrity *at the same time*. We designated these file utilities as able to protect themselves when dealing with untrusted data such that their functionalities can be preserved.

Instead of trusting these utilities to consume any untrusted data, Spif can further reduce the set of files by relying on the "implicit-explicit" technique described in Section 6.1. When users invoke a command, data files are specified as input arguments[2].

A side effect of making these utilities as trusted is that their outputs have high integrity labels. This is not desirable for applications like `cp` and `tar` as integrity labels on original files are lost. We solved this problem by setting appropriate flags to preserve the integrity information. This is relatively easy as the integrity information is encoded as group ownership in SPIF.

8.6 Evaluation of SPIF

In this section, we evaluated the complexity, compatibility, usability, security and performance of SPIF.

SPIF shares the same code complexity as in the dual-sandboxing architecture (Section 4.5.2).

8.6.1 Preserving Functionality of Code

We performed compatibility testing with about 100 applications shown in Figure 8.1a on Ubuntu. 70 of them were chosen randomly, the rest were hand-picked to include some widely used applications. Figure 8.1b shows a list of 35 unmodified applications that can run successfully at high- and low-integrity in SPIF on Windows. We used them to perform basic tasks. These applications span a wide range of categories: document readers, editors, web browsers, email clients, media players, media editors, maps, and communication software.

Benign mode

As expected, all the applications running as benign processes worked perfectly when given benign inputs.

To use these applications with untrusted inputs, we first ran them with an explicit `uudo` command or from untrusted shell (bash on Ubuntu or cmd on Windows). In this mode, they all worked as expected. When used in this mode, most applications modified their preference files, and our approach for redirecting them worked as expected.

We then used these applications with untrusted inputs, but without an explicit `uudo`. In this case, our `uudo` inference procedure was used,

[2]When globbing is used in shell command, the shell process will expand it to the set of file names matching the pattern.

Document Readers	Adobe Reader, dhelp, dissy, dwdiff, evince, F-spot, FoxitReader, Geegle-gps, jparse, naturaldocs, nfoview, pdf2ps, webmagick
Document Processor	Audacity, Abiword, cdcover, eclipse, ewipe, gambas2, gedit, GIMP, Gnumeric, gwyddion, Inkscape, labplot, lyx, OpenOffice, Pitivi, pyroom, R Studio, scidavis, Scite, texmaker, tkgate, wxmaxima
Games	asc, gbrainy, Kiki-the-nano-bot, luola, OpenTTD, SimuTrans, SuperTux, supertuxkart, Tumiki-fighters, wesnoth, xdemineur, xtux
Internet	cbm, evolution, dailystrips, Firefox, flickcurl, gnome-rdp, httrack, jdresolve, kadu, lynx, Opera, rdiff, scp, SeaMonkey, subdownloader, Thunderbird, Transmission, wbox, xchat
Media	aqualung, banshee, mplayer, rhythmbox, totem, vlc
Shell-like	bochs, csh, gnu-smalltalk, regina, swipl
Other	apoo, arbtt, cassbeam, clustalx, dvdrip, expect, gdpc, glaurung, googleearth, gpscorrelate-gui, grass, gscan2pdf, jpilot, kiki, otp, qmtest, symlinks, tar, tkdesk, treil, VisualBoyAdvance, w2do, wmmon, xeji, xtrkcad, z88

(a) Software tested on Ubuntu

Readers	Adobe Reader, MuPDF
Document Processor	MS Office, OpenOffice, Kingsoft Office, Notepad 2, Notepad++, CppCheck, gVim, AklelPad, IniTranslator, KompoZer
Internet	Internet Explorer, Firefox, Chrome, Calavera UpLoader, CCProxy, Skype, Tor + Tor Browser, Thunderbird
Media	Photoshop CC, Picasa, GIMP, WinAmp, Total Video Player, VLC, Picasa, Light Alloy, Windows Media Player, SMPlayer, QuickTime
Other	Virtual Magnifying Class, Database Browser, Google Earth, Celestia

(b) Software tested on Windows

Figure 8.1: Software ran successfully in Spif

and it worked without a hitch when benign applications were started using a double-click or a "open-with" dialog on the file manager `nautilus` or `Windowsexplorer`. The inference procedure also worked well with simple command-lines without pipelines and redirection.

One example that the technique did not handle well was when double-checking to open an untrusted image file on Windows. The default viewer is the running explorer process itself, which is a benign process and hence cannot read the untrusted file. Users have to open the image file with another editor (e.g., MS Paint) or have a default program other than the Explorer such that SPIF can perform the `uudo` inference.

Untrusted mode

All of the software shown in Figure 8.1 worked without any problems or perceptible differences. We discuss our experience further for each category shown in Figure 8.1.

Document Readers All of the document readers behave the same when they are used to view benign files. In addition, they can open untrusted files without any issues. They can perform "save as" operations to create new files with untrusted label.

Games By default, we connect untrusted applications as untrusted X-clients, which are restricted from accessing some advanced features of the X-server such as the OpenGL GLX extensions. As a result, only 8 out of 12 games worked correctly in this mode. However, all 12 applications worked correctly when we used (the some what slower) approach of using a nested X-server (`Xephyr`).

Editors/Office/Document Processors These applications typically open files in read/write mode. However, since our system does not permit untrusted processes to modify benign files, attempts to open benign files would be denied. Most applications handle this denial gracefully: they open the file in read-only mode, with an appropriate message to the user, or prompt the user to create a writable copy before editing it.

Internet This category includes web browsers, email clients, instant messengers, file transfer tools, remote desktop clients, and information retrieval applications. All these applications worked well when run as untrusted processes. Files downloaded by applications are correctly labeled as untrusted. Any application opening these downloaded files will hence be run in untrusted mode, ensuring that they cannot damage system integrity.

Media Player These are music or video players. Their functions are similar to document readers, i.e., they open their input files in read-only mode. Hence, they do not experience any security violations. For media editors, they behave more like document processors. They create new media files rather than modifying the original files.

Shell-like application This category includes shells or program interpreters that can be executed interactively like a shell. Once started in untrusted mode, all the subsequent program executions will automatically be performed in untrusted mode.

Other Programs We tested a system resource monitor (`wmmon`), file manager
(`tkdesk`), some personal assistant applications (`jpilot`, `w2do`, `arbtt`), `googleearth` and some other applications. We also tested a number of specialized applications: molecular dynamic simulation (`gdpc`), DNA sequence alignment (`clustalx`), antenna ray tracing (`cassbeam`), program testing (`qmtest`, `expect`), computer-aided design (`xtrkcad`) and an x86 emulator (`bochs`). While we are not confident that we have fully explored all the features of these applications, we did observe the same behavior in our tests in benign as well as untrusted modes. The only problem experienced was with the application `gpscorrelate-gui`, which did not handle permission denial (to write a benign file) gracefully, and crashed.

Overall

Reading both high and low integrity files. Applications that only read, but not modify files can always start as low-integrity, so that they can consume both high and low integrity files.

Editing both high and low integrity files. SPIF does not allow a process to edit files of different integrity simultaneously as this can compromise the high-integrity files. However, SPIF allows files to be edited in different processes— edit high-integrity files in high-integrity processes, and edit low-integrity files in low-integrity processes. As these processes are running as different users, different instances of the same application can run simultaneously in SPIF.

Low-integrity processes writing high-integrity files. Applications like OpenOffice maintain runtime information in user profile directories. Applications expect these files to be both readable and writable— otherwise they will simply fail to start and crash. Having these files as high-integrity would prevent low-integrity processes from being usable. Letting these files become low-integrity would break availability of high-integrity processes.

SPIF shadows accesses to these files inside user-profile directories, hence high- and low-integrity processes can both run without significant usability issues. One problem is that profiles for high and low integrity sessions are isolated. There is no safe way to automatically merge the shadowed files together.

On Ubuntu, files that are being shadowed are all "." entries. Some of them are cache file, some of them are preference/history files (`.viminfo`, `.pulse − cookie`, `deluge/ui.conf`, `gtkfilechooser.ini`, `vlcrc`, `.recently − used.xbel`) or cache files. As for Windows, shadowing is primarily applied to preference files. Specifically, SPIF applies shadowing to files in `%USER PROFILE%\AppData`, `HKEY_CURRENT_USER` and files in all hidden directories. None of them corresponds to data files and deleting the redirected storage does not result in any significant usability issues.

8.6.2 Usage experience

Secure-by-design solutions frequently end up requiring considerable changes to applications as well as the user experience. We walk through several usage scenarios to demonstrate that our techniques generally do not get in the way of users, and are highly compatible with existing software. Here are some scenarios to illustrate the usability of SPIF, as well as how SPIF preserved the normal user experience.

Watching a movie We opened a movie torrent from an untrusted website. `Firefox` downloaded the file to the temporary directory and labeled it as untrusted. The default BitTorrent client, `Transmission`, was invoked as untrusted to start downloading the movie into the Download directory. Once the download completed, we double-clicked the movie to view it. `vlc` was started as untrusted to play the movie. Realizing that the movie had no subtitles, we located `subdownloader` for downloading subtitles. Since our installer considers Ubuntu's universe repository as untrusted, the application was installed as untrusted, and hence operated only in untrusted mode. We searched and found a match. Clicking on the match resulted in launching an untrusted `Firefox` instance. We went back to `subdownloader` to download the subtitle, and then loaded this file into `vlc` to continue watching the movie.

Compiling programs from students Some students submit their programming assignments. Teaching assistants for the course need to download their projects, extract them, compile them and execute the binaries in order to grade the assignments. In this experiment, we considered an attack that creates a backdoor by appending ssh key to `authorized_keys` so that a malicious student can break into TA's machine later.

With protection from SPIF, when the TA received the submission as an attachment, it was marked untrusted. As the code was unpacked, compiled and run, this "untrusted" label stayed with it. So, when the code tried to append a public key, it was stopped.

Resume template We downloaded a compressed resume template from the Internet. When we double clicked on the tgz file, `FileRoller`, the default archive manager started automatically as untrusted because the file was labeled as untrusted by `Firefox`. We extracted the files to Documents directory. We then opened the file with `texmaker` by selecting "Open With", since `texmaker` was not the default handler for tex file. `texmaker` was started as untrusted and we started editing the file. We then compiled the latex file and viewed the dvi document with `evince` by clicking on the "View DVI" button in `texmaker`. We then viewed pdf and `AdobeReader` was automatically invoked as untrusted. The document was rendered properly.

Stock charting and analysis We wanted to study trend of a stock and we searched the Internet about how to analyze. We came across a tutorial on an unknown website with a `R` script. We installed `R` and downloaded the script. When we started `R`, we found that it is a command line environment and is not so user-friendly for beginners. We then installed `RStudio`, a front-end for `R`, from a deb file we found on another unknown website. Our installer installed `RStudio` as untrusted because `Firefox` labeled the deb file as untrusted. After we started `RStudio`, we loaded the script and realized that it required several `R` libraries. We installed the missing `R` libraries. These libraries were installed in a shadow directory since `R` implicitly accessed the library directory. After installing the libraries, we generated a graph. We saved the graph in the Pictures directory, and edited the graph with `GIMP`.

Summary The protection offered by Spif had allowed us to download and run arbitrary software. Applications were started in the right mode automatically and user did not have to think about security.

While security failures occur from time to time, our efforts to ensure application transparency bore fruit: applications handled failures gracefully if not transparently. For instance, if an untrusted editor was used to open a benign file, it would first attempt to open the file in read/write mode, which would be denied. Then it simply opens the file in read-only mode, and the user does not experience a difference unless she tries to edit the file.

8.6.3 Experience with malicious software

Spif is also effective in stopping malware from compromising the system. Here we present some scenarios involving stealthy attacks that are stopped by our system.

Real world malware on Ubuntu Malware can enter systems during installation of untrusted software or via data downloads. As secure installation is not our focus, we assumed that attacks during installation are prevented by systems like [Sun et al., 2008a] or the system presented in Section 10 and untrusted files are labeled properly.

We tested our system with malware available on [Packet Storm, 2015] for Ubuntu and [Offensive Security, 2014] for Windows. On Ubuntu, these malware were mainly rootkits: patched system utilities like `ps` and `ls`, kernel modules, and `LD_PRELOAD` based libraries. Specific packages tested include: `JynxKit`, `ark`, `BalaurRootkit`, `Dica`, and `Flea`. All of them tried to overwrite benign (indeed, root-owned) files, and were hence stopped.

`KBeast` (Kernel Beast) requires tricking root process to load a kernel module. The benign sandbox prevents root processes from loading the kernel module since the module is labeled as untrusted.

Real world exploit on Ubuntu We tested an Adobe Flash Player exploit (CVE-2008-5499) on Ubuntu, which allows remote attackers to execute arbitrary code via a crafted SWF file. If the browser is simply trusted to be free of vulnerabilities, then this attacks would obviously succeed. Our approach was based on treating the web-site as untrusted, and opening it using an untrusted instance of the browser. In this case, the payload executed, but its actions were contained by the untrusted sandbox. In particular, it could not damage system integrity.

Simulated targeted attacks on Ubuntu We also simulated a targeted attack via compromising a document viewer on Ubuntu. A user received a targeted attack email from an attacker, which contained a PDF that can compromise the viewer. When the user downloaded the file, the email client labeled the attachment as untrusted automatically since the sender cannot be verified. Our system, however, did not prevent the user from using the document. User could still save the file along with other files.

When she opened the file, the document viewer got compromised. On an unprotected system, the attacker controlled viewer then dumped a hidden malicious library and modified the `.bashrc` file to setup environment variable `LD_PRELOAD` such that the malicious library would be injected into all processes the user invoked from shell. Worse, if the user has administrative privileges, the viewer can also create an alias on `sudo`, such that a rootkit would be installed silently when user performs an administrative action.

Although the viewer still got compromised under SPIF, the user was not inconvenienced: while she could view the document, modification attempts on `.bashrc` were denied, and hence malware attempts to subvert and/or infect the system were thwarted.

CVE/OSVDB-ID	Application	Attack Vector
2014-0568	Adobe Reader	Code
CVE-2010-2568	Windows Explorer (Stuxnet)	Data (lnk)
2014-4114/113140	Windows (Sandworm)	Data (ppsx)
104141	Calavera UpLoader	Preference (dat)
100619	Total Video Player	Preference (ini)
2013-6874/100346	Light Alloy	Data (m3u)
2013-3934	Kingsoft Office Writer	Data (wps)
102205	CCProxy	Preference (ini)
2013-4694/94740	WinAmp	Preference (ini)
2014-2013/102340	MuPDF	Data (xps)

Figure 8.2: Exploits defended by SPIF on Windows

8.6.4 Real world exploit on Windows

There are far more malware available on Windows. We evaluated the security of SPIF against malware from Exploit-DB [Offensive Security, 2014] on Windows XP, 7 and 8.1. We selected all local exploits targeting Windows platform, mostly released between January and October of 2014. Since these exploits work on specific versions of software, we only included malware that "worked" on our testbed, and their results were easy to verify. Figure 8.2 summarizes the CVE/OSVDB-ID, vulnerable applications, and the attack vectors. We classify attacks into three types: data input attacks, preference/configuration file attacks, and code attacks.

Note that by design, SPIF protects high-integrity processes against all these attacks. Since high-integrity processes cannot open low-integrity files, only low-integrity applications can input any of the malware-related files. In other words, *attackers can only compromise low-integrity processes.* Moreover, there is no mechanism for low-integrity processes to "escalate their privilege" to become high-integrity processes. Note that since low-integrity processes can only modify files within the shadow directory, they cannot affect any user or system files. For this reason, SPIF *stopped all of the attacks shown in Figure 8.2.*

Both data and preference/configuration file attacks concern inputs to applications. When applications fail to sanitize malicious inputs, attackers can exploit vulnerabilities and take control of the applications. Data input attacks involve day-to-day files like documents (e.g., wps, ppsx, xps). They can be exploited by simply tricking users to open files. On the other hand, attacks using preference/configuration files are typically hidden from users, and are trickier to exploit directly. These exploits are often chained together with code attacks to carry out multi-steps attacks to circumvent sandboxes.

Code attacks correspond to instances where the attacker is already able to execute code but with limited privileges, e.g., inside a restrictive sandbox. For instance, in the Adobe Reader exploit [Fisher, 2014], it is assumed that an attacker has already compromised the sandboxed worker process. Although attackers cannot run code outside of the sandbox, they can exploit a vulnerability in the broker process. Specifically, the attack exploited the worker-broker IPC interface — the broker process only enforced policies by resolving the first level NTFS junction. A compromised worker can use a chain of junctions to bypass the sandbox policy and write arbitrary file to the file system with the broker permissions. Since the broker ran with user privilege, attackers could therefore escape the sandbox and modify any user files. Spif ran both the broker and worker as untrusted processes. As a result, the attack could only create or modify low-integrity files, which means that any subsequent uses of these files were also confined by the untrusted sandbox.

Spif stopped Stuxnet [Falliere et al., 2011] by preventing the lnk vulnerability from being triggered. Since the lnk file is of low-integrity, Spif prevented Windows Explorer from loading it, and hence stopped Windows Explorer from loading any untrusted DLLs.

We also tested the Microsoft Windows OLE Package Manager Code Execution vulnerability, called Sandworm [Ward, 2014]. It was exploited in the wild in October 2014. When users view a malicious PowerPoint file, OLE package manager can be exploited to modify a registry in `HKLM`, which subsequently triggers a payload to run as system-administrator. Spif ran PowerPoint as low-integrity when it opened the untrusted file. The exploit was stopped as the low-integrity process does not have permissions to modify the system registry.

The most common technique used to exploit the remaining applications was an SEH buffer overflow. The upload preference file `uploadpref.dat` of Calavera UpLoader and `Setting.ini` of Total Video Player were modified so that when the applications ran, the shell-code specified in the files would be executed. Similarly, SEH buffer overflow can also be triggered via data input, e.g., using a multimedia playlist (.m3u) for Light Alloy or a word document (.wps) for Kingsoft Office Writer. Other common techniques include integer overflow (used in `CCProxy.ini` for CCProxy) and stack overflow (triggered when MuPDF parsed a crafted xps file or when WinAmp parsed a directory name with invalid length). In the absence of Spif, these applications ran with user's privileges, and hence the attackers could abuse user's privileges, e.g., to make the malware run persistently across reboots.

Although preference files are specific to applications, there exists no permission control to prevent other applications from modifying them. Spif makes sure that preference files of high-integrity applications cannot be modified by any low-integrity subject. This protects benign processes from being exploited, and hence attackers cannot abuse user privileges. On the other hand, Spif does not prevent low-integrity instances of the applications from consuming low-integrity preference or data files. While attackers could exploit low-integrity processes, they only had privileges of the low-integrity user. Furthermore, all attackers' actions were tracked and confined by the low-integrity sandbox.

8.6.5 Real world malware on Windows

One of the advantage of implementing the defense on Windows is that we can evaluate the defense against a wide range of malware available in the wild.

We downloaded more than 30000 files from malwr.com [Claudio nex Guarnieri and Alessandro jekil Tanasi, 2015], a website where anyone can submit files for dynamic malware analysis. Malwr.com relies on Cuckoo [Cuckoo Foundation, 2015], an open source automated malware analysis tool to analyze submissions. Cuckoo works by running the files inside VMs and monitors the behaviors of the processes. As some malware may only exhibit malicious behaviors only when there are human interactions (e.g., mouse move events), Cuckoo generates these events. Cuckoo relies on injecting libraries to monitor processes.

Out of the 30000 files downloaded from malwr.com, 14366 of them are executables. We focus our automated testing on executables only. To evaluate the effectiveness of Spif, we modified Cuckoo. We prepared two groups of Windows XP SP2 VMs, one group without Spif and one group with Spif protection. Since Spif's library works at a lower level than Cuckoo's, events observed by Cuckoo library were redirected and shadowed; Cuckoo can report user file being modified, yet Spif transparently shadowed the modifications. We therefore do not rely on the monitoring facility in Cuckoo. To obtain the exact changes made by processes during executions, our system dumps the entire system registry tree and generates md5 for all files in the system before and after the running of each of the samples. By comparing the snapshots, our system can detect changes made by the samples. By comparing the snapshots generated from protected VM and unprotected VM, we can determine if Spif is effective in stopping the sample.

9579 samples showed no changes when running in both unprotected and protected VMs, of which 6746 were marked as malware by Virus Total. The reason why these samples do not exhibit any observable behavior is likely because of missing system dependency or library in our VM, or they detected virtualized environment and refuse to run. The remaining 4787 samples showed changes in our testbed.

2538 of the 4787 samples modified the system registry entries, of which 1272 of them modified entries to automatically start whenever the system boots. Spif stopped these attempt because untrusted processes do not have privileges to modify system objects. 1096 samples modified user registry entries or files so that they can start whenever the user login to the system. Spif redirected these changes to the untrusted users' registry entries or files. As untrusted user does not login to the system via the login windows, these entries will not be used. 8 samples modified the security zone mapping. Again, Spif redirected these changes and hence have no effect to R. 1246 samples attempted to create files or executables in `C:\Windows` or `C:\Program Files`. Spif's policy does not allow untrusted processes to create files in these directories, as they can compromise all other untrusted processes. 23 samples modified existing executables. Spif does not allow existing system executables to be modified by untrusted processes. For user executables, untrusted processes can modify them but R is not affected because of shadowing. 12 of the samples modified a large number of user files on the system. These include randsome malware which encrypts user files. Spif does not allow them to modify user files because these are considered as benign data file. Modifications to non-data files would be shadowed by Spif. As a result, benign programs and data files are not affected. 1819 samples created new executable in non-system locations. Spif redirected the creation to redirected directories. These new files also carry untrusted labels so that they can only run as untrusted processes. Any effect from these untrusted processes will be tracked by Spif.

8.6.6 Performance

A practical malware defense should have low overheads. We present both micro and macro-benchmarks of Spif. All performance evaluation results

	Simple syscall	Simple read	Simple write	Simple stat	Simple fstat	Simple open/ close	Select on 10 fd's	Select on 100 fd's
Unprotected (μs)	0.80	0.88	0.86	2.14	0.98	3.94	1.03	2.10
`platform` (overhead)	6.18%	5.98%	5.60%	5.34%	4.88%	232%	5.13%	2.36%
remote (overhead)	6.39%	6.10%	5.85%	174%	4.97%	183%	4.91%	1.89%

	Pipe latency	Process fork + exit	Process fork+ execve	Process fork + /bin/sh
Unprotected (μs)	74.57	438	1128	2498
`platform` (overhead)	1.79%	55.04%	173%	152%
remote (overhead)	2.03%	54.63%	149%	134%

Figure 8.3: `lmbench` performance overhead on Ubuntu

were obtained on Ubuntu 10.04 and Windows 8.1. (Performance does not vary much across different versions of Windows.)

Micro-benchmark Figure 8.3 shows the performance of `lmbench` micro-benchmark. `stat` has large overhead for untrusted processes because we are consolidate `stat` into `fstatat` for untrusted processes. The overhead for `open`/`close` is particularly high because of the implicit/explicit tracking. The behavior on fork-related calls are likely to be because of the use of `fork` instead of `vfork`.

Figure 8.4 shows the `SPEC2006` benchmark overheads on Ubuntu and Windows. The overhead is less than 1% for CPU intensive operations. This is to be expected, as the overhead of SPIF will be proportional to the number of intercepted system calls or Windows API calls, and SPEC benchmarks make very few of these.

Macro-benchmark Figure 8.5 shows the overhead of `openssl` and `Firefox` when compared with unprotected systems on Ubuntu. We obtained the statistics using `speed` option in `openssl`. As for `Firefox`, we used `pageloader` addon [Mozilla, 2015] to measure the page load time. Pages from top 1200 Alexa sites were fetched locally such that overheads due to networking is eliminated. The overhead on `openssl` benchmark is negligible. The average overhead for `Firefox` on Ubuntu is less than 5%. We did the same experiment on Windows for the top 1000 Alexa sites. The overheads for benign Firefox and untrusted Firefox on Windows are 3.32% and 3.62% respectively. Figure 8.6 shows the correlation between unprotected page load time with

	Unprotected Time (s)	Benign Overhead	Untrusted Overhead
400.perlbench	575.8	-0.18%	0.10%
401.bzip2	841.8	0.23%	-0.38%
403.gcc	541.2	-1.99%	0.82%
429.mcf	699.0	-0.86%	-1.06%
445.gobmk	693.2	-0.02%	-0.02%
456.hmmer	982.7	0.36%	-0.13%
458.sjeng	933.8	0.49%	0.51%
462.libquantum	995.4	-0.17%	0.33%
464.h264ref	1243.3	0.21%	-0.27%
471.omnetpp	573.0	0.07%	-0.24%
473.astar	734.2	-0.46%	-0.79%
433.milc	882.5	0.85%	-2.66%
444.namd	841.5	0.11%	0.13%
Average		-0.10%	-0.28%

(a) SPEC2006 on Ubuntu

	Unprotected Time (s)	Benign Overhead	Untrusted Overhead
401.bzip2	1785.9	-0.33%	0.26%
429.mcf	716.4	-1.69%	-0.96%
433.milc	3314.1	1.15%	-0.53%
445.gobmk	1094.9	0.26%	-0.08%
450.soplex	1108.0	0.58%	2.34%
456.hmmer	2386.2	0.02%	0.13%
458.sjeng	1442.5	-0.25%	0.20%
470.lbm	1203.0	-1.51%	-0.32%
471.omnetpp	750.9	0.96%	1.83%
482.sphinx3	2653.6	-2.55%	-3.45%
Average		-0.34%	-0.06%

(b) SPEC2006 on Windows

Figure 8.4: Overhead in SPEC2006, `ref` input size

	Benign		Untrusted	
	Overhead	σ	Overhead	σ
`openssl`	0.01%	1.43%	-0.06%	0.70%
`Firefox`	2.61%	4.57%	4.42%	5.14%

Figure 8.5: Runtime overhead for Firefox and OpenSSL on Ubuntu.

protected benign Firefox and untrusted Firefox.

We also evaluated SPIF with Postmark [Katcher, 1997], a file I/O intensive benchmark. To better evaluate the system for Windows environment, we tuned the parameters to model files on a Windows 8.1 system. There were 193475 files on the system. The average file size is 299907 bytes, and the median is a much smaller 5632 bytes. We selected 3 size ranges based on this information: small (500 bytes to 5KB), medium (5KB to 300KB), and large (300KB to 3MB) bytes. Each test creates, reads, writes and deletes files repeatedly for about 5 minutes. We ran the tests multiple times and the average is presented in Figure 8.7. There are three columns for each file size, showing (a) the base runtime obtained on a system that does not have SPIF, (b) the overhead when the benchmark is run as a high-integrity process, and (c) the overhead when it is run as a low-integrity process. As expected, the system shows higher overhead for small files. This is because there are more frequent file creation and deletion operations that are intercepted by SPIF. For larger files, relatively more time is spent on reads and writes, which are not intercepted by SPIF.

Figure 8.8 shows the latency for some GUI programs on Ubuntu. We measured the time between starting and closing the applications without using them. While there are some latencies, the overall user experiences were not affected when using the applications.

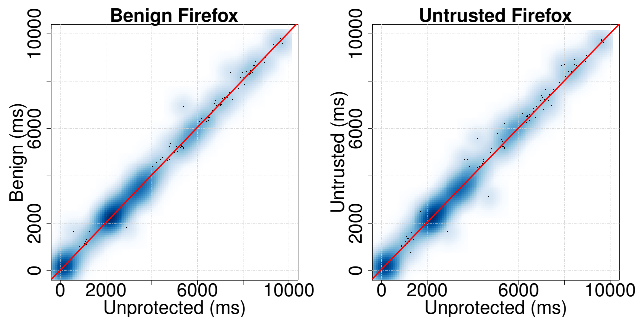


Figure 8.6: Firefox page load time correlation on Windows

File Size	500B to 5KB		
Operations	Base	Benign	Untrusted
Files Created per Second	351.14	-5.02%	-10.45%
File Read per Second	350.14	-5.18%	-10.59%
File Appended per Second	344.79	-5.19%	-10.58%
File Deleted per Second	350.21	-5.17%	-10.57%
Total Transaction Time (s)	285.36	6.53%	12.38%
File Size	5KB to 300KB		
Operations	Base	Benign	Untrusted
Files Created per Second	68.00	-2.79%	-2.02%
File Read per Second	67.64	-3.02%	-2.34%
File Appended per Second	67.64	-3.02%	-2.61%
File Deleted per Second	67.86	-3.03%	-2.00%
Total Transaction Time (s)	367.29	3.05%	4.58%
File Size	300KB to 3MB		
Operations	Base	Benign	Untrusted
Files Created per Second	8.00	-1.25%	-1.56%
File Read per Second	7.60	-3.95%	-1.97%
File Appended per Second	8.00	-2.50%	-2.34%
File Deleted per Second	8.00	-1.25%	-2.34%
Total Transaction Time (s)	308.67	1.27%	-0.62%

Figure 8.7: Postmark overhead for high and low integrity processes on Windows

	Unprotected	Benign	Untrusted
	Time (s)	Overhead	Overhead
eclipse	6.16	1.99%	10.23%
evolution	2.44	2.44%	5.04%
F-spot	1.61	2.11%	6.80%
Firefox	1.32	3.24%	10.08%
gedit	0.82	5.02%	6.09%
gimp	3.63	1.90%	4.32%
soffice	1.56	0.33%	7.08%

Figure 8.8: Latency for starting and closing GUI programs on Ubuntu

8.6.7 Discussion

Alternative choices for enforcement

Spif could use WIM labels instead of userids for provenance tracking and policy enforcement. WIM enforces a no-write-up policy that not only prevents a low-integrity process from writing to high-integrity files, but also to processes. Although WIM does not enforce no-read-down, we can achieve it in a co-operative manner using an utility library, the same way how Spif achieves it now.

With userids, Spif gets more flexibility and functionality by using DAC permissions to limit the access of untrusted processes. For instance, files that can be read by low-integrity applications can be fine-tuned using the DAC mechanism. Moreover, Spif can be easily generalized to support the notion of groups of untrusted applications, each group running with a different userid, and with a different set of restrictions on the files they can read or write. Achieving this kind of flexibility would be difficult if WIM labels were used instead of userids.

On the positive side, WIM can provide better protection on desktop/window system related attacks. The transition to lower-integrity is also automatic when a process executes a lower-integrity image, whereas this functionality is currently implemented in our utility library. For added protection, one could combine the two mechanisms — this is a topic of ongoing research.

Limitations

Our WinAPI interception relies on the `AppInit_DLLs` mechanism, which does not kick in until the first GUI program runs. Furthermore, libraries loaded during the process initialization stage are not intercepted. This means that if a library used by a benign application is somehow replaced by a low-integrity version, a malicious library could be silently loaded into a high-integrity process. Our current defense relies on the inability of untrusted applications to replace a high-integrity file, but subtle attacks may be possible where an application loads a DLL from the current directory *if* the DLL is present, but if the DLL is not found, it starts normally. A better solution is to develop a kernel driver to enforce a no-read-down policy on file loads.

Our prototype does not consider IPC that takes place through COM and Windows messages. COM supports ACL, so it may be easy to handle.

Our prototype does not support untrusted software whose installation phase needs administrative privileges. If we enforce the no-read-down policy, the installation won't proceed. If we waive it, then malicious software will run without any confinement, and can damage system integrity. Techniques for secure software installation [Sun et al., 2008a] can be applied to solve this problem, but will need to be implemented for Windows.

Other architectural/implementation vulnerabilities

Attacks on U_H Policies on untrusted processes are enforced using the well-defined, well-studied DAC mechanisms. Relaxation over the strict policy dictated by the DAC mechanisms are provided via the use of helper process U_H. Communications between untrusted processes and U_H use UNIX-domain sockets. This narrow communication interface exposed to untrusted processes have only a small attack surface. Same as other benign processes, U_H cannot be `ptraced` or receive signals from untrusted processes. Furthermore, U_H runs with user's privilege, but not administrative privileges.

Vulnerabilities in trusted programs Trusted programs are trust-confined. They are only trusted to consume some specified untrusted files. Opening of other untrusted files will still be rejected. Our approach reduces the attack surface to those interfaces where we incorrectly chose to trust. Trusted programs are those that can execute in limited mutual trust mode. They are only trusted to consume some specified untrusted files. Opening of other untrusted files will still be rejected. Our approach reduces the attack surface to those interfaces where we incorrectly chose to trust.

Trusted programs may also need to label files based on origins. Labeling errors due to misplaced trust. For instance, we may incorrectly trust a software provider and mark their code as benign or label an untrusted file downloaded from untrusted source as benign. The only defense in this regard is to be conservative: only trust those sources that are indisputably trustworthy.

Labeling errors Another related problem is labeling errors due to misplaced trust. For instance, we may incorrectly trust a software provider and mark their code as benign or label a untrusted file downloaded from untrusted source as benign. The only defense in this regard is to be conservative: only trust those sources that are indisputably trustworthy.

8.7 Generalizing to multiple principals

While partitioning origins into benign and untrusted is effective in protecting the system integrity against malware, there are several drawbacks for considering only two principals. First of all, this coarse-grained partitioning groups all potentially malicious resources into the same category. A single malicious file can ruin all untrusted but non-malicious resources. This is particularly problematic since most resources cannot be placed in the benign category. The second problem is that putting all resources in the same level does not preserve security boundary. Naturally, files from both origin A and B can be untrusted to the system; yet, information from A should be isolated from B. The two-principal model cannot support this isolation. In this section, we discuss how to generalize SPIF to support multiple untrusted principals, i.e., provenance tracking for more than 2 sources.

We also generalize SPIF to support confidentiality. Secret information such as SSH authorization keys, cookies, and password files contains authentication information that attackers can use to gain access to resources. Protecting integrity alone does not prevent such attacks. Protecting confidentiality is necessary to provide complete protection.

We extends the notion of principal in OSes to incorporate not only local user but also network provenance information. This is also a generalization to the same-origin-policy in web browsers and the app model in Android, where only provenance information of app (web app or Android app) is considered.

OSes only support mutually untrusted relationships between principals. Information is not shared between principals by-default, but principals are free to share information voluntarily. In the previous section, SPIF introduced a mandatory unidirectional trust relationship between two principals. By considering multiple principals and confidentiality, SPIF can capture a more general notion of trust and model trust-hierarchy in various systems. For example, Android allows one application to "invoke" another application for code reuse; Web browsers isolate code and data so that code from one origin can only access data and interact with code from the same origin. At the same time, browsers also support using third-party scripts as libraries. Our extension to SPIF can simulate existing trust models such as Android and Bubbles [Tiwari et al., 2012].

8.7.1 Implementation and evaluation

Setup: We have modified Spif to support multiple principals. We created additional users on the system to represent new principals. From the OS perspective, these new principals are no different than regular users.

Each principal has a set of principals that it trusts with integrity and confidentiality. In our experiment, we have created four principals: *platform*, *word_processor*, *pdf_reader*, and *untrusted*. *word_processor* corresponds to a principal which owns the Microsoft Word. *pdf_reader* corresponds to a principal that owns Adobe Reader. *platform* trusts no other principal; Both *word_processor* and *pdf_reader* trust *platform*; *untrusted* trusts all of the other principals.

Figure 8.9 specifies the policy used for all the principals. Note that *word_processor* and *pdf_reader* are mutually untrusted to each other. No information is permitted to flow between them.

Policy enforcement: Spif relies on OS DAC permissions to enforce object rules. If principal A does not allow principal B to perform an operation on A's objects, Spif will protect A's objects with DAC permissions by denying B from performing the operation. On Windows, DAC permissions are encoded using ACL which supports both positive and negative ACL entries. Spif encodes the entire set of object rules using ACL. On Linux, since ACL is not widely supported, Spif uses the 9-bit DAC permission to encode a default deny policy. It is up to the subjects to request the operation using a helper process. The helper process will then decide whether to permit the operation based on the object rules.

The enforcement of subject rules is based on the observation that these rules are defined by the subject principal itself to restrict information flowing into and out of the principal. As a result, the subject has no interest in bypassing subject rules. Therefore, Spif enforces subject rules using library interception.

For enforcing `allow read` and `allow write` rules, whenever a subject opens files of different principal for reading or writing, Spif checks if the permissions are allowed by the policy. The enforcement of the object rules such as `allow all read` and `allow all write` are enforced by DAC permission using ACL. For systems like Unix that do not use ACL, Spif relies on U_H to mediate accesses to other principals' objects and enforce object policies for other principals.

For enforcing principal transition rules, the transition rule `allow transition to` is checked when the subject tries to transition into another principal before executing an executable. The other transition rule `allow transition from` is enforced by `uudo` to check if the subject principal has the permission to execute the object.

Shadowing and redirection: Every principal (except `platform`) has its own shadowing and redirection directory. Shadowing resolves the write object conflicts using copy-on-write, where an executing principal does not have permission to modify an object. Redirection provides user an illusion that files are located in the same directory; yet, they are located in different redirection directories.

The purpose of redirection is to protect applications that are not compatible with SPIF-library. Since SPIF enforces subject rules using SPIF-library, applications that do not load the SPIF-library will not be protected against accidental consumption of untrusted resources. By partitioning resources based on principals, applications that are not compatible with SPIF will remain protected. Note that since SPIF does not rely on SPIF-library to enforce object policies. Applications that do not load SPIF-library still cannot bypass the object rules, which are enforced using DAC permissions.

When a principal lists a directory, SPIF creates an unified view for the principal by combining all the redirected directories. For example, *untrusted* principal listing user's home directory will contain the results not only from its own directory, but also from *word_processor*, *pdf_reader*, and *platform*. Since redirection is only applied to data files, SPIF ensures that file names cannot collide by rejecting the creation of files with the same name.

On the other hand, configuration and preference files use a shadowing/copy-on-write semantics, with read-only copies from trusted principals. Modifications to the files would result in shadowing in the principal's own shadowing directory. For instance, when the *untrusted* principal runs Microsoft Office, SPIF allows the *untrusted* Microsoft Office process to read but not modify *word_processor*'s preference files. When these files are updated, SPIF transparently shadows the changes in *untrusted*'s shadow directories.

Performance In term of performance, we observed no performance difference for supporting MultiP in SPIF. This is because SPIF relies on OS DAC permission to label and enforce policies. The labeling and enforcement are independent of the number of provenance labels, as they are simply users from the OS perspective.

The principal resolution algorithm kicks in for every calls to `CreateProcess` or `exec`. Since there are only a limited number of process creation, the algorithm imposes negligible overheads.

Another factor that could affect the performance is to check if an operation involving multiple principal is allowed or not. The procedure for retrieving the object principal has been there even in the two-provenance case. The only additional work for supporting MultiP is to consult the policy. The policy is loaded in the memory and SPIF rely on hash table to perform the lookup. Hence, it imposes negligible overheads.

Transition between principals: SPIF is an early downgrading system (Section 5), principal transitioning therefore happens at `exec` or `CreateProcess` time. For every `exec` or `CreateProcess` call, SPIF runs the principal resolution algorithm (Section 7.6.1) to decide what can be the executing principal. At the time of invocation, SPIF knows the invoker principal, the code owner principal, and some of the data owner principals.

In the experiment, a *platform* process starting a Microsoft Office application will automatically transition into *word_processor* because the invoker is *platform* and the code owner is *word_processor*. No rule in the *platform*'s policy would allow *platform* to execute code from any other principals. Hence, the transition rule (Rule 4 and 12) kicks in to transition the *platform* process to *word_processor* to allow Microsoft Office to run.

All the documents created by the Office applications will therefore belong to *word_processor*. Users can save the document as a PDF file on the desktop. The PDF document still belongs to *word_processor*. However, when users double click on the PDF document belonging to the *word_processor* principal, SPIF will need to design again which principal to run the code.

The data file belongs to *word_processor*, yet the application belongs to *pdf_reader*. Neither the *word_processor* nor the *pdf_reader* trusts information from the other. Hence, the PDF reader cannot run as any of the two. The domain transition rules (Rules 11, 15, and 6) allow transitioning to *untrusted*. Rule 7 and 10 also allows *untrusted* to read the PDF file from *word_processor*. Rule 7 and 14 also allows *untrusted* to read the config and preference files from *word_processor*. As such, the PDF reader runs as *untrusted*. An exploit in the PDF document would therefore cannot compromise the *pdf_reader* principal.

8.7.2 Confidential policy samples

Figure 8.10 shows a list of files that contain browser confidential information. To protect these confidential information, the *platform* principal can include these files as confidential, or the system administrator can create a *browser* principal and only let *browser* to read them.

Figure 8.11 shows a list of files that contain FileZilla FTP site confidential information. We can similarly protect them by setting them as confidential.

```
1.  confidential = {.ssh/*, ...}
2.  deny     all         read         confidential
3.  allow    all         read, exec   *
4.  allow    transition  to           any
```

(a) *platform*'s policy

```
5.  allow  all                read, exec   *
6.  allow  transition         to, from     any
7.  allow  read, write, exec  *   from     platform, word_processor, pdf_reader
```

(b) *untrusted*'s policy

```
8.  confidential = {HKEY_CURRENT_USER\...\Office\...\Identity}
9.  deny     all                read         confidential
10. allow    all                read, exec   *
11. allow    transition         to           any
12. allow    transition         from         platform
13. allow    read, write, exec  *            from         platform
```

(c) *word_processor*'s policy

```
14. allow    all                read, exec   *
15. allow    transition         to           any
16. allow    transition         from         platform
17. allow    read, write, exec  *            from         platform
```

(d) *pdf_reader*'s policy

Figure 8.9: Policy for multi-principal system

```
.*\Mozilla\Firefox\Profiles\.*\.default\signons\.sqlite$
.*\Mozilla\Firefox\Profiles\.*\.default\secmod\.db$
.*\Mozilla\Firefox\Profiles\.*\.default\cert8\.db$
.*\Mozilla\Firefox\Profiles\.*\.default\key3\.db$
.*\History\History\.IE5\index\.dat$
.*\Temporary Internet Files\Content\.IE5\index\.dat$
.*\Application Data\Google\Chrome\.*
```

Figure 8.10: Files for protecting browser private information

```
.*\FileZilla\sitemanager\.xml$
.*\FileZilla\recentservers\.xml$
```

Figure 8.11: Files for protecting FileZilla private information

Chapter 9

SRFD: Integrity Protection with Dynamic Downgrading

The weaknesses of userid-based DAC has prompted a resurgence of interest in mandatory access control (MAC) [Loscocco and Smalley, 2001b, Biba, 1977, Krohn et al., 2007, Sun et al., 2008b, Zeldovich et al., 2006, Efstathopoulos et al., 2005, Ubuntu, 2015, Li et al., 2007, Mao et al., 2011, Sze and Sekar, 2013, Fraser, 2000]. Information-flow approaches such as the Biba model [Biba, 1977] are particularly attractive in the context of malware threats, as they can prevent low-integrity (untrusted and potentially malicious) data or code from ever influencing high-integrity data or applications. It not only prevents malware from directly corrupting important system files, but also stops indirect attacks that operate by corrupting some intermediate data consumed by other high integrity processes that can update important system files.

A drawback of the Biba model is that its strict separation between high and low-integrity objects and subjects, which impacts its usability. Consider a utility application such as a word-processor that needs to operate on both high and low integrity files. It would be necessary to have two versions of every such application, one for operating on high-integrity files and another for low-integrity files. It is cumbersome to install and maintain two versions of every application. Worse, a user needs to be careful in selecting the correct version of an application for each task — choosing a high-integrity version of an application for processing low-integrity files (or vice-versa) will lead to security failures and/or application crashes.

The low-water-mark policy [Biba, 1977] can avoid these drawbacks of the strict policy by permitting subject integrity to be downgraded at runtime. In particular, this policy allows applications to be invoked with high integrity, and the integrity level to be downgraded if the application subsequently reads a low integrity object. Fraser [Fraser, 2000] argues eloquently why low-water-mark policy provides significantly better compatibility with existing software as compared to the strict model. However, prior to this project, the low-water-mark policy was not very popular because of the *self-revocation* problem [Fraser, 2000]. Specifically, consider a subject that has already opened a high integrity file for writing. If this subject subsequently opens a low integrity file for reading, it is downgraded. At this point, the subject cannot be permitted to write the high integrity file any more. Applications expect and handle security failures when opening files, but once opened, they assume that subsequent read and write operations will not fail. When this assumption is invalidated, applications may malfunction or crash.

In this chapter, we implemented a more general solution to the self-revocation problem in all cases based on our SRFD policy. Our implementation of SRFD on Ubuntu Linux 13.10 is fast, incurring a maximum overhead under 6% and average overhead below 2% across several macro-benchmarks. The evaluation also demonstrates that SRFD provides very good compatibility, while thwarting malware attacks.

9.1 Implementation

We implemented SRFD based on the mechanism described in Section 3 with the design in Section 5.5 .

We present an implementation and experimental evaluation of SRFD on Ubuntu Linux 13.10. Our experimental evaluation shows that our implementation is fast, incurring a maximum overhead under 6% and average overhead below 2% across several macro-benchmarks. The evaluation also demonstrates that SRFD has very good compatibility while thwarting malware attacks.

LSM hooks are used to enforce information flow policies, perform dynamic downgrading, track and maintain `min_lbl` constraints. Our implementation also uses an user-level component to perform some usability enhancing features such as notifying users when a process is downgraded and shadowing accesses to preference files for low-integrity processes. By maintaining separate preference files for high and low-integrity processes, SRFD prevents processes from downgrading automatically due to consuming low integrity preference files. Note that these features do not allow a process to bypass kernel enforcement.

9.1.1 Abstraction mapping: Subjects, Objects, and Handles

SRFD maps threads to subjects. Threads of the same process belong to the same subject group. Within the kernel, subjects are identified using `task_structs`. Since LSM does not have hooks to track process creation directly, our prototype relies on `cred_*` hooks instead. For each subject group, SRFD maintains information such as integrity level and a list of handles.

Objects are mapped into `inodes` in the kernel. Our implementation maintains and updates object-related information, including labels, handles associated with each object, and constraints. We use LSM hooks on inodes for creating objects on demand, and deallocating objects when they are no longer needed. For file objects, integrity labels are stored on the disk persistently using extended attributes.

Handles are similar to file descriptors but represent an unidirectional information flow between exactly one subject and one object. SRFD relies on LSM hooks such as `file_open`, `inode_permission` and `d_instantiate` to maintain handles. When an object is associated with a subject (as a result of a file open, pipe or socket creation), the object will be attached to the subject via at least one handle. When the association is broken, e.g., due to a `close` operation, the corresponding handle is destroyed.

9.1.2 Constraint propagation

When a subject A opens a file O for writing (or a socket connection with another process), constraints from the file (or target process) have to be propagated in the inverse direction of information flow, as described in Section 3.3.3. The open operation is permitted if the invariants regarding `current_lbl` and `min_lbl` can be satisfied after this propagation.

Note that constraint propagation can involve circular dependencies as illustrated in Figure 3.1. To deal with cycles, SRFD uses a fix-point algorithm for constraint propagation. To detect a fix-point, SRFD stores the previous value of `min_lbl` in a variable called `last_min_lbl`. It then updates the value of `min_lbl` of A to be the maximum of `last_min_lbl` and the label of the file O. If `min_lbl`$(A) =$ `last_min_lbl`(A), then a fix-point has been reached, and our algorithm stops. If not, then the same process is used to propagate the new value of A's `min_lbl` to each of the subjects $S_1, \ldots, S_n$ that output to A, and the process continues. If any of the propagation steps fail because it results in a `min_lbl` exceeding the value of `current_lbl`, then the `open` operation is denied, and the values of `min_lbl` restored.

The same fix-point algorithm is used even if A performs a `close` rather than an `open`. The only difference is that instead of computing the maximum of A's `min_lbl` and that of the new object being opened, we recompute `min_lbl` as the maximum of the labels of all the currently open write handles of A. However, in the presence of cycles, this simple algorithm will not always compute the least fix-point. For this reason, our algorithm will retry constraint propagation from scratch before denying an open request. Note that (a) this retry step is unnecessary if no `close` operations have taken place since the last retry, and (b) constraint propagation itself is unnecessary for processes that are already at low-integrity.

LSM has no hooks on `close` operation: SRFD is not notified when a process closes a file. As a result, SRFD may have stale information regarding files opened. SRFD solves this problem by walking through the file descriptor table to prune out outdated handles when recomputing constraints. SRFD optimizes this by recomputing the constraints only when the current constraints cannot be satisfied.

9.1.3 Tracking subjects

Processes inherit a lot of rights from their parents, e.g., ability to write to a file. SRFD needs to be aware of these inherited rights to protect against self-revocation of these rights.

When a new process is created, SRFD duplicates the book-keeping information associated with the parent to the child. This automatically captures the communication between parent and child that happen using mechanisms such as pipes. The most common use of pipes occur in the context of shell processes, where the parent first creates a pipe with a readable-end and a writable-end. It then creates two child processes. At this point, the parent and children can all read and write from the pipes, so there is cyclic dependency between them. As a result, any constraint propagation will result in all three processes having the same `min_lbl`. However, in the next step, parent shell will close the two ends of the pipe, and then the first child will close the readable end of the pipe, while the second child will close the writable end of the pipe. After these close operations, there can be no flow between the children and the parent shell. Moreover, no information can flow from the second child to the first child. All of this is handled by our constraint propagation algorithm, which will correctly allow the second child to be downgraded (if necessary) without having to downgrade the first child or the parent.

9.1.4 Limitations

Our current prototype does not enforce its policies on operations relating to capabilities, file mount points, signals, message queue, and semaphores. In particular, low-integrity processes performing these operations are not restricted. We also simply denies lower integrity processes to ptrace on higher integrity processes. We have left out these aspects since our experiments did not make use of these system calls. A complete implementation should also mediate these operations by propagating labels.

For sockets, our prototype handles Unix domain sockets because the two ends of the socket connection are within the control of the OS. For sockets in the Internet domain, their other end is typically outside the control of the OS. Hence SRFD does not attempt to enforce any policies on such Internet sockets.

9.2 Performance

We evaluate the performance of SRFD using micro- as well as macro-benchmarks. All the evaluations are performed on a Ubuntu 13.10 VMware virtual machine allocated with one VCPU AMD Opteron Processor 4228 HE (2.8GHz) and 1GB RAM.

As a micro-benchmark, we use `lmbench`, which measures the overhead for making individual system calls. Figure 9.1 shows the overheads of our system for different classes of system calls. The overheads are modest: the geometric mean is about 12%, and the arithmetic mean is 16%. Note that if we exclude open and close, which are typically less frequent than other calls such as read/write, the overheads are much smaller — less than 5%.

	Simple syscall	Simple read	Simple write	Simple stat	Simple open/ close	Select 10 fd's	Select 500 fd's
unprotected	0.375	0.477	0.517	1.104	2.591	0.624	8.935
protected	0.376	0.526	0.580	1.122	5.867	0.624	8.958
Overhead (%)	0.09%	10.28%	12.15%	1.62%	126%	-0.1%	0.26%

	Pipe latency	AF_UNIX latency	Process fork+ exit	Process fork+ /bin/sh -c	**Geometric mean**
unprotected	12.854	8.812	235.4	1830	
protected	13.994	9.785	249.8	1963	
Overhead (%)	8.87%	11.04%	6.08%	7.27%	**12%**

Figure 9.1: SRFD lmbench Performance overhead

It is natural for `open` and `close` to have higher overheads because of constraint propagation, but that does not explain a doubling of execution time. It occurs in our prototype because LSM does not have hooks for `close`, and as a result, our implementation has to walk through the list of open file descriptors while propagating constraints. In contrast, because there can be no failures on read and write, no additional checking is needed, and the only work is to blindly copy `current_lbl` from the source to destination.

Micro-benchmarks help to understand and explain the overheads of kernel-based defenses such as ours, but they tend to overestimate the overheads because most applications spend only a minority of their time in the kernel. Macro-benchmarks are better at estimating overheads experienced by real users in practice. For this reason, we used several macro-benchmarks, including the CPU-intensive SPEC 2006 and openssl, file-system intensive `Postmark`, and commonly used programs such as browsers and software builds.

From Figures 9.2 and 9.3, it is clear that overheads on CPU-intensive programs such as SPEC and openssl are negligible — the overheads are below measurement errors/noise.

Package builds, which represent a combination of CPU and I/O load, show a slightly higher overhead of 1% to 3%. Specifically, our benchmark built Debian Linux packages for `coreutils` and `am-utils` from source code. Another mixed load consists of Firefox, whose overhead was measured using `pageloader`, a benchmarking tool from Mozilla. Top 3000 Alexa sites were pre-fetched in this experiment so as to eliminate the effects of network latency. (If this was not done, then the overheads will be even smaller.) The overhead

	Unprotected Time (s)	Protected Overhead
400.perlbench	554.41	-0.21%
401.bzip2	772.29	0.03%
403.gcc	505.47	0.01%
429.mcf	709.06	0.02%
445.gobmk	673.06	0.05%
456.hmmer	712.94	-0.13%
458.sjeng	865.29	-0.23%
462.libquantum	1032.35	-0.23%
464.h264ref	1159.41	-0.05%
471.omnetpp	543.24	0.27%
473.astar	738.29	0.16%
433.milc	875.47	-0.14%
444.namd	764.47	-0.09%
Average		0.04%

Figure 9.2: SPEC2006 Overhead for SRFD, `ref` input size

	Protected Overhead
Openssl	-0.08%
dpkg -b coreutils	2.93%
dpkg -b am-utils	1.22%
Firefox	4.89%
Postmark	5.74%

Figure 9.3: Overhead on other benchmarks for SRFD

experienced was 5%.

Finally, the I/O-intensive `Postmark` was configured to create 500 files with size between 500 bytes and 500 Kbytes. The overhead reported was 6%.

9.2.1 User Experience

Our work is motivated by a continuing trend in sophisticated and adaptive malware attacks, and our desire to develop a principled defenses against them. Existing approaches rely on techniques such as sandboxing a few key applications such as browsers and email readers that have the most exposure to malware. While sandboxing these applications can prevent some attacks, e.g., those that try to mount a code injection attack on an email reader (or other document viewers invoked by a browser), more sophisticated attacks can often get around these defenses. For instance, users may save a document on their desktop, and subsequently open it with their favorite document editor/viewer application. Since the application is typically not sandboxed in this usage scenario, the attack can succeed. In contrast, an information-flow based approach would mark such files as low-integrity, and regardless of the number of applications that process them, or how many intermediate steps they go through, untrusted files will always be operated on by low-integrity processes. Since such processes can only output low-integrity files, and cannot modify high-integrity files or interfere with high-integrity subjects, their attempts to compromise system integrity will continue to fail.

Although these theoretical benefits of information-flow based integrity protection are well-known, these techniques have not found widespread use on modern operating systems as they often pose compatibility challenges. In this section, we walk through several illustrative and common usage scenarios to demonstrate that SRFD can work well on contemporary operating system distributions, without posing major compatibility problems. Naturally, our focus will be on illustrating features specific to SRFD, as opposed to information-flow based techniques in general.

In these scenarios, we assume that the default OS installation consists of only high-integrity files; and that low-integrity files enter the system when the system begins to be used, and new low-integrity files are created by low-integrity subjects. We assume that browsers and email readers are run as low-integrity processes.

Self-revocations involving files, pipelines and sockets

The scenarios discussed here illustrate the benefits of accurate information-flow dependency tracking in SRFD, and how that permits us to be more functional as compared to previous approaches (specifically, LOMAC [Fraser, 2000]), while avoiding self-revocation.

One of the challenges in SRFD is to track communications between processes. This can be nontrivial when a deep pipeline is involved. Consider the command:

```
cat lowI | grep... | sed | ... | sort | uniq ⟩⟩ highI
```

It is necessary to propagate labels across the pipeline to ensure that information from low-integrity file `lowI` is prevented from contaminating a high-integrity file `highI`. Opportunities for self-revocation abound, especially if the shell opens `highI` before `cat` gets a chance to open `lowI`. Even otherwise, self-revocation is possible since intermediate commands such as `grep` may begin execution as high-integrity processes, and then be prevented from reading their input pipes, or they may be downgraded and prevented from writing on their output pipes. LOMAC [Fraser, 2000] avoids self-revocation on pipes by downgrading process groups at a time — in this case, all processes in the pipeline will be part of the same process group.

SRFD accurately captures information flow dependencies between the processes in the pipeline, and can avoid self-revocation while preserving usability. In particular, depending on the order in which processes are scheduled, `cat` may be permitted to downgrade. In this case, SRFD will deny the open operation on `highI`. Alternatively, if `highI` is opened first, SRFD will deny `cat`'s attempt to open `lowI`.

Another example that illustrates the strength of SRFD is:

```
cat high1 | tee high2 | lowP
```

where `lowP` is a low-integrity utility program. SRFD will run this pipeline successfully: both `cat` and `tee` will be remain at high-integrity, and be able to output to high-integrity file `high2`, while `lowP` will run at low-integrity. LOMAC requires all processes in the pipeline to be at the same level, and hence cannot run this.

SRFD protects sockets, and can avoid self-revocation on processes that make use of these features. When a server program has a high-integrity file opened for writing, SRFD will deny connections from a low-integrity client, as the establishment of such a connection would violate the constraints on `min_lbl`. Moreover, any client that is already connected to such a high-integrity server will be prevented from opening a low-integrity file, or connecting to any other low-integrity process. LOMAC will experience self-revocation.

Commonly used applications

We implemented SRFD on a Ubuntu 13.10 desktop system. This system runs a large number of applications and services, including a number of daemons, X-server, GNOME desktop environment, and so on. All these applications work with SRFD, but this is unsurprising: in our tests, these applications did not access low-integrity files, and so SRFD does not constrain them in any way.

In the same manner, applications that don't modify high-integrity files will run without any problems, as SRFD imposes no constraints on them. Most complex applications can be run this way — for instance, we run web browsers and email readers in this mode.

Most command-line programs can run as high or low-integrity without any problems. Common utilities such as `tar`, `gzip`, `make`, compilers, and linkers can be run without any problems on low-integrity files. Composing these command line applications using pipelines works as described in the preceding section. Thus, we focus the rest of this section on more complex GUI applications that need to access a combination of low and high-integrity files.

Document viewers Document viewers such as evince and Acrobat Reader can be used in SRFD without any issues. These programs can be used to open high and low-integrity documents simultaneously. However, once the viewer has opened a low-integrity file, it will not be able to overwrite a high-integrity file.

Editors GUI editors (e.g., gedit, OpenOffice, GIMP) impose additional challenges for dynamic downgrading systems like SRFD. When users select files to edit using file selection dialogs, applications tend to open every file to generate a preview, regardless of the integrity of the files. When users open a directory containing low-integrity files, the editors will automatically be downgraded to low-integrity even if the users did not intend to open low-integrity files.

To prevent editors from downgraded accidentally, we can allow editors to be downgraded only when demanded by users. We can rely on the "implicit-explicit" mechanism suggested in Section 6.1 to identify file accesses that are requested explicitly by users, and only allow editors to be downgraded on opening these files. SRFD can deny opening low-integrity files implicitly.

Media Editors We consider media editors (e.g., f-spot and audacity) separately because they usually do not modify the original media files directly.

Instead, they edit copies of the media files. As a result, these media editors can be used without usability issues.

9.2.2 Defense against malware

We downloaded a rootkit `ark` from [Packet Storm, 2015]. The tar file was labeled as low-integrity when downloaded into the system by a web browser. The user then untars the file by invoking `tar`. SRFD started `tar` as a high-integrity process, with `current_lbl` $= Hi$, `min_lbl` $= Lo$ because it has no constraints on its output files and it has not been contaminated with any low-integrity information. `tar` started by loading libraries like `ld.so.cache` and `libc` $-$ `2.17.so`. The `tar` process was then downgraded to low-integrity when reading the rootkit tar file. `tar` process then spawned `gzip` as low-integrity to decompress the file. After decompressing, the `tar` process continued to untar. All of the new files created are automatically labeled as low-integrity.

With these integrity labels in place, SRFD can easily preserve system integrity. Specifically, system directories are labeled as high-integrity and hence rootkits cannot be placed in the system directories. It is possible for users to accidentally invoke these rootkits by placing them in some user-specific search paths. SRFD protects the system integrity by downgrading processes when these rootkits are executed or used, including executions by root processes. Hence, when a user process executes a low-integrity binary or loads a low-integrity library, SRFD downgrades the process and prevent the process from damaging system integrity.

SRFD also intercepts LSM hooks related to kernel modules. Low-integrity kernel modules cannot be loaded even by root processes.

9.3 Discussion

The LOMAC project [Fraser, 2000] does not attempt to solve the self-revocation problem in its entirety, but focuses on two common instances that involve pipes and shared memory abstractions. Pipes are particularly nasty, because downgrading of one process in a pipeline can prevent it from writing to the pipe, which in turn will cause the next process in the pipeline to fail because it does not get any input. LOMAC avoids this problem by permitting pipe communications only within a UNIX process group, and ensuring that all processes within this group are at the same level. This notion of a group is further extended to include all processes that share memory. Since all processes within a group are at the same level at all times, there is no need to restrict communication among them, and hence pipes and shared memory operations don't ever have to be denied. Unfortunately, self-revocation problem still remains when dealing with files, as well as other IPC mechanisms such as sockets.

Both UMIP [Li et al., 2007] and IFEDAC [Mao et al., 2011] adopt the LD model and do not constrain high-integrity processes. High-integrity process can therefore be downgraded accidentally due to the consumption of low-integrity input. This can cause all its future accesses to be denied, including writes to files that were opened before consuming low-integrity input.

Flume [Krohn et al., 2007] uses the notion of endpoints and rules to enforce endpoint safety. File endpoints have immutable labels. This implicitly constrains the labels of processes — processes cannot downgrade (e.g., by acquiring low-integrity label) as this will violate the file endpoint safety constraints. This prevents self-revocation. However, Flume only solved self-revocation involving single process. It does not constrain downgrading for processes that are connected via IPC. When these processes downgrade themselves to consume low-integrity data, the IPC endpoint safety can no longer be satisfied. As a result, messages will get dropped silently. SRFD addresses this problem by propagating constraints across all IPC-connected processes. Furthermore, Flume does not consider file close operations. Once a file is opened, the file endpoint constraint remains throughout the process lifetime. As a result, the process can never downgrade themselves once they opened a high integrity file for writing, even after closing the files. While SRFD also relies on LSM hooks, SRFD handle for file close operations by searching the file description table. Hence, SRFD allows processes to be downgraded after closing high-integrity files.

Chapter 10

Software Installation: A Case Study for Deferred Enforcement

In this chapter, we discuss an important application of Delayed Enforcement, namely software installation. We introduce SwInst to secure the software installation process. SwInst is built based on Spif and transaction, a delayed enforcement mechanism. Transaction allows deferring the decision on whether to accept changes made by installers. SwInst allows system administrators to customize policies about acceptable changes. Installations that do not violate the policies will be committed automatically, otherwise, the system state is reverted as if the installation process has never taken place.

One of the important criteria for any provenance tracking system is the proper object labeling upon object creation. The two main ways to introduce new files into systems are through browser downloads and software installation. We have already described how Spif leverages browser add-ons and Security Zone to label file downloads properly. In this chapter, we discuss how to leverage the provenance tracking capability of Spif to label programs created during software installation and hence securing the installation process.

We applied a delayed enforcement approach towards securing software installation.

Software installation itself poses a significant challenge to malware defense; not only because it introduces new programs into the system, but also because the installation process itself involves running untrusted code with administrative privileges. This would allow malware to shutdown any user-level and kernel-level protection mechanism, and malware can also install themselves persistently into the system at firmware levels [Hudson, 2015].

Desktop OSes vendors realized the problem and have attempted to address the problem partially. Instead of limiting what an installer process can do, OS vendors specifically focus on protecting their system files against tampering. Microsoft uses digital signatures to protect some of the system binaries; Apple supports System Integrity Protection [Apple Inc., 2015b] to allow only Apple-signed processes to update some of the system files. These mechanisms do not attempt to protect applications or the user environment because there is no way for the OSes to distinguish if it is user's intention to modify the applications and the user environment.

Modern OSes such as Android, iOS, Windows 10 and OS X adopt container based model for installing applications. Each app lives in its own directory which contains all the libraries and other dependencies the app needs. Apps are also independent of each other. Installation and uninstallation of the app is as simple as creating and deleting the app directories. Unfortunately, most desktop applications (e.g., Microsoft Office, Photoshop, Adobe Reader, Firefox) do not run as apps; furthermore, a complete separation of apps also limits app functionalities. Modern desktop OSes such as Windows 10 and OS X therefore still support the traditional software installation— i.e., let the installers to do whatever they want to install the applications.

Users expect software installers to install programs into system directories and configure the system in order to work properly, and therefore they are willing to grant installers administrative privileges. Users do not expect installers to compromise the integrity of their systems; however, there exists no mechanism to make sure that the installers will only perform what they are supposed to do. OSes only enforce the bare minimum policies to protect themselves, yet leaving users no way to confine but to trust the installers.

Software installations are attractive for both malware and PUPs (Potentially Unwanted Programs). For malware writers, instead of finding exploits to compromise programs to run their payloads, software installation allows them to run arbitrary code directly with administrative privileges. They can create registry entries or files so that they will be persistent across system reboot. They may also modify browser settings in an unwanted way. For software distributors, by distributing and installing PUPs inattentively along with their software, the distributors can make extra profits.

Our goal is to secure the software installation process. Our system, SwInst, works by dividing software packages into different trust levels and imposes different restrictions for different packages. Intuitively, users are more willing to give packages from trustworthy sources more privileges than packages from less trustworthy sources. In this paper, we evaluate the possibility of restricting privileges on installers. This is challenging as installers usually run with administrative privileges without any confinement.

Specifically, our contributions are:

- Designed and implemented SwInst that secures the software installation process for Debian OSes
- Evaluated SwInst by testing installation of over 17500 packages

10.1 Existing installation approaches

One way to install software is via invoking `make install` or running software installers directly, where users download the software and run scripts or binaries provided by the packages (e.g., in the form of `Makefile`, `install.sh`, `.msi`, or `.pkg`). All desktop OSes support this type of software installation. Users usually need to make sure that the system has met all the dependencies requirements of the software.

A more common approach towards software installation on Linux is via package managers, which helps resolving dependencies. Package manager front-ends (such as `apt-get` or Ubuntu Software Center) help users to find and retrieve packages from pre-configured repositories. Users can also download pre-compiled installation packages (in the form of deb or rpm files) manually. These installation packages contain dependency information that can be used to check for conflicts and dependency. The package manager back-ends (e.g., dpkg) will perform the actual installation by extracting files from packages into the file system. Although this may seem less dangerous than the make install approach, scripts from the packages (pre- and post-installation scripts) do execute directly.

SwInst supports both installation methods. We focus our discussion on the package manager based installation as it also involves running scripts provided directly from the packages.

10.2 Difficulties in securing the installation process

Installers run with administrator's privileges; however, there exists no mechanism to limit trust on the installers. A malicious installer can:

- replace existing files with rootkits
- mark a file as root-setuid binary such that it can escalate to root at a later time
- create a new user with uid 0
- make a protected file as world-writable
- control another running process

Even if we limit installers from performing the above operations, they may still need to modify some files legitimately. How do we make sure that files are modified in a legitimate way?

We propose SwInst, a system to safe-guard the installation process on unmodified installers. SwInst also makes it easy to develop policy to safely install untrusted applications.

10.3 Threat Model

We assumed that packages can be partitioned into benign and untrusted—only packages coming from untrusted origins may compromise the system integrity. SwInst works by imposing restrictions on untrusted package installers to protect system integrity. SwInst therefore cannot support arbitrary untrusted packages installation. SwInst is designed to support most untrusted package installation automatically. With the 17,161 packages randomly selected from the Ubuntu repositories, 87% of them could be installed without violating SwInst's policy.

Software installation involves not only creating new files, but also modifying existing files. For example, database of installed packages will be updated during the installation, a new helper program may register itself as capable of opening certain files, or a program may want to create a new non-system

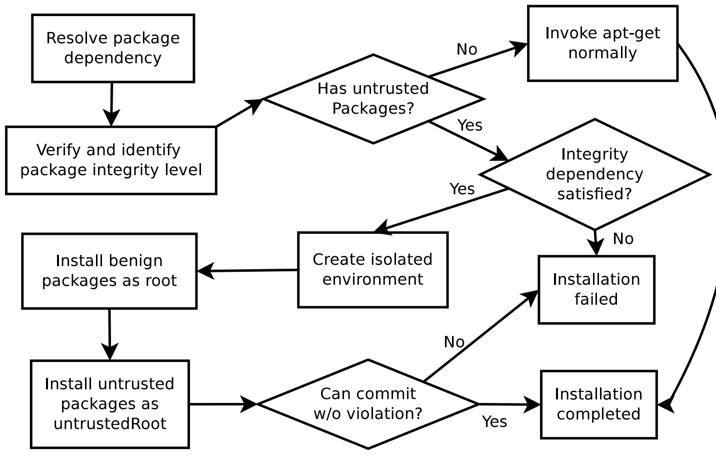

Resolve package dependency
Verify and identify package integrity level
Has untrusted Packages?
No
Yes
Invoke apt-get normally
Integrity dependency satisfied?
Yes
No
Create isolated environment
Install benign packages as root
Installation failed
Install untrusted packages as untrustedRoot
Can commit w/o violation?
No
Yes
Installation completed

Figure 10.1: Software installation flow chart in SwInst

user. SwInst needs to make sure that all the changes are safe. Instead of writing policies focusing on what changes are acceptable, SwInst focuses on securing how the changes are commenced, i.e., the chain of processes that has resulted in the file changes. Policies are specified in terms of program invocation. This provides high-level reasoning about why some modifications are safe, and has significantly reduced the policy development efforts. Since the policies are developed based on invoking existing system utilities, SwInst requires untrusted installers to modify files using system utilities rather than editing the files directly.

10.4 System Design

SwInst protects against untrusted installers by isolating the untrusted installation processes. When the installation is completed, SwInst analyzes if the modifications are acceptable, and commits the changes back to the system only if so. This commit-based design is more powerful than sandboxing approach. In this section, we describe how SwInst handles the pre-installation phase, installation phase and post-installation phase. Figure 10.1 shows an overview flow of the installation. We discuss each of the step in this section.

10.4.1 Handling dependency

Before installation, SwInst prioritizes the package installation order. On unprotected system, all packages specified by users, as well as dependent packages, will be installed at the same time with administrator privilege. SwInst divides the installation into two phases by first installing benign packages, and then the untrusted packages.

SwInst allows users to mark certain repositories as untrusted. Our implementation on Ubuntu provides a wrapper on `apt-get`. Users interact with the wrapper the same as the the original `apt-get`. Upon receiving requests to install new packages, the wrapper first resolves the dependency and download the packages. At the same time, the wrapper identifies the integrity level of the packages based on matching package checksums with the repository database. After identifying the integrity-levels, the wrapper installs the benign packages and then the untrusted packages.

SwInst ensures that the integrity dependency is satisfied before the installation, i.e., benign packages do not depend on any untrusted package. Otherwise, SwInst will deny the installation.

10.4.2 Isolating installation

Since installation scripts assume that they can modify, create, and remove any file with administrative privilege, revoking such capabilities can break installation. SwInst protects the system against untrusted installers by one-way isolation, which virtualizes the resources for installers. Apart from isolation at the file system level, SwInst also needs to protect other processes running in the system. Running untrusted code with root privileges can allow untrusted code to control any other processes. Instead of running untrusted installers unconfined, SwInst applies both chroot and setuid-jail to restrict file system and IPC accesses.

SwInst runs untrusted installers as a new, unprivileged user *untrustedRoot*. No other process runs with *untrustedRoot*. To isolate modifications to the file system, SwInst creates a copy-on-write (COW) file system and chroots untrusted installers inside it. Files owned by root cannot be modified even in the COW file system. To allows *untrustedRoot* to modify root-owned files inside the COW, a root helper process running outside COW will handle file open requests from *untrustedRoot*. The helper will open files in the COW directory in writable mode. Before passing the file descriptor to the

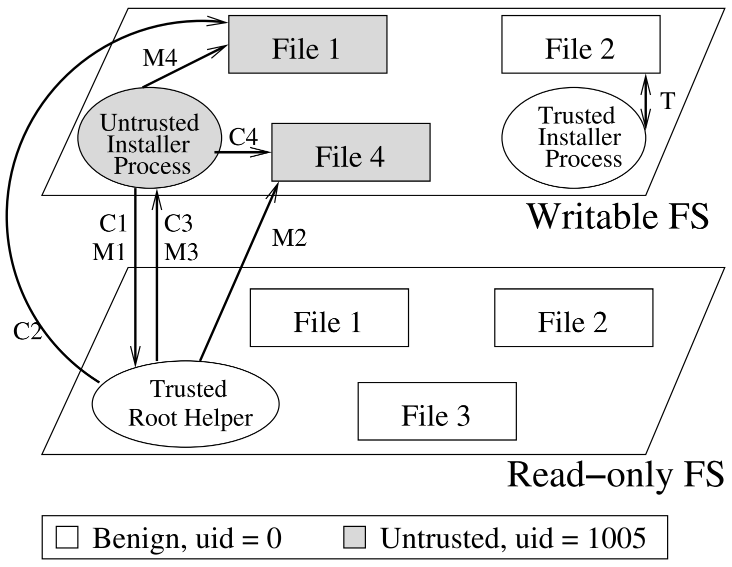


Figure 10.2: SwInst architecture
SwInst architecture relies on COW filesystem, chroot and setuid jail

installer, the root helper will change the ownership of the file to belong to *untrustedRoot*. This allows the analyzer to know that this file could have been modified by *untrustedRoot in arbitrary way.*

SwInst requires no modification to installers. SwInst injects a library into installers to change some of the system call behaviors. For example, when file opens failed, the library transparently requests the helper process to open the files, and then replace the error with the file handles returned from the helper process. SwInst does not rely on the library to enforce any policy, but to facilitates the installation process within the isolation environment. The actual policy is enforced when analyzing the changes.

Figure 10.2 shows the overall architecture for SwInst. Edges labeled with C and M correspond to file creation and modification by untrusted process respectively. The trusted root helper helps the untrusted installer to handle file creation and modification, and the helper will label these files as untrusted at the same time. Edge T corresponds to the file access by trusted installer. This access is not mediated. We describe more in Section 10.4.4 about trusted programs.

10.4.3 Committing changes

The COW file system shadows all the files changed, created and removed during an installation. SwInst can decide if the installation is safe or not by simply examining the files in the shadow directory. SwInst enforces a policy that no existing benign file should be modified to untrusted.

However, limiting changes to only file creation would break most installations; at least the package database files will be updated. SwInst supports two strategies to validate if modifications to files are safe: file-based and invocation-based. File-based verifications compare the difference between the pre-installation and post-installation version of the file. Rules are defined on what changes are acceptable for each file. Invocation-based verifications do not rely on file contents, but on how the changes to the files are produced. SwInst defined a set of invocation rules suggesting files changed by certain programs are always safe to commit. SwInst ensures that files are only modified by programs satisfying the safe condition. This is done by ensuring that invocation parameters and the invocation environment are safe. Since these files cannot be modified by *untrustedRoot* in arbitrary way, these files are not owned by *untrustedRoot*. The analyzer does not need to perform any validation to the file contents. We will discuss more how SwInst guarantee the integrity of the invocation environment in later section.

Apart from verifying the safety of the file content, the analyzer will also make sure that files modified in the rootfs have not been modified since the start of the installation. If the files in the rootfs are modified, the installation process might have used an out-dated copy during the installation. Overwriting the rootfs files could result in inconsistency.

If the changes can be committed back to the rootfs, the analyzer will simply move the files from the shadow storage to the rootfs. Otherwise, the changes will be discarded and the analyzer will report to the user why the installation failed.

Files may be deleted during an installation. In COW filesystem, the underlying filesystem usually creates *whiteout* files in the writable branch to represent deleted files. SwInst cannot encode how a file is deleted using permission (as in the file modification case). SwInst solved this problem by introducing a directory for *authorized deletion*. If a file is not deleted in arbitrary way, i.e., deleted only under specific circumstances, the deletion will result in an entry in the authorized deletion directory. SwInst relies on the portable integrity protection system to ensure the integrity of the trusted processes and the authorized deletion directory. We will discuss more in the following subsection.

10.4.4 Invoking trusted programs by untrusted installers

Files can be modified arbitrarily during the installation. Developing policies to capture safe modifications for each file would require a lot of efforts. We observe that file modifications are consequences of invocation of some commands. We therefore propose verifying safety at the time of invocation of the commands. For example, by verifying that `useradd` was invoked with a non-zero userid, SwInst does not need to verify if `/etc/passwd` was modified safely.

However, programs invoking with the right parameters does not automatically guarantee the files modified are safe to commit. Files can be modified by other processes in addition to the intended process. Furthermore, untrusted installers could have compromised the execution environment for the intended processes. It is therefore important to protect the execution of trusted programs.

SwInst enforces information flow policy using Spif to protect the integrity of the execution environment of trusted programs. Instead of running trusted programs as *untrustedRoot*, SwInst runs trusted programs as *root*. This prevents untrusted installers from injecting code into processes running trusted programs. SwInst also protects the execution environment with a default policy in Spif to ensure that trusted processes cannot consume files created/modified by *untrustedRoot*. This is because reading untrusted files could compromise the integrity of trusted processes. Similar to Spif, SwInst achieves this by injecting a library into trusted processes— the library monitors every file access and ensures that all the files accessed are not owned by *untrustedRoot*. SwInst also protects system libraries and the SwInst-library by denying untrusted installers from modifying the libraries, despite they are inside a COW environment.

SwInst transitions from untrusted processes to trusted processes in two steps. In the first step, untrusted processes will check if the `exec` parameters satisfy the conditions for running as trusted processes. If so, a setuid-to-root program will be executed with the existing parameters passed as arguments. In the second step, the setuid-to-root program will validate the parameters again since the checking performed by untrusted processes cannot be trusted. Furthermore, the setuid program will also make sure that the pro-

```
"/usr/man/*",
"/usr/share/man/*",
"/usr/local/man/*",
"/usr/local/share/man/*",
"/usr/X11R6/man/*",
"/opt/man/*",
```

Figure 10.3: Untrusted files that SwInst trusts `mandb` for reading

gram image is safe. As for environment variables, existing systems already protect them for setuid programs— loader automatically ignore the environment variables such as `LD_PRELOAD` and `LD_LIBRARY_PATH` when executing setuid programs. Hence, SwInst does not have to worry about malicious environment. Trusted programs cannot consume arbitrary untrusted files. SwInst only allows trusted programs to read files located at specific directories. SwInst considers `mandb` as a trusted program. Figure 10.3 shows the policy of the files that SwInst trusts `mandb` to read.

By default, new programs executed by any trusted process will run as untrusted processes. This is achieved by calling `setuid` to *untrustedRoot* before invoking `exec`. For scripts, the trust would be inheritable to child processes.

10.5 Policy development

Policy development in SwInst is based on training. Policies are updated when installation of seemingly safe untrusted packages has resulted in violation. To facilitate the policy development process, SwInst traces installers running inside the confined environment. SwInst produces not only files that violated the policy, but also how the violation was resulted.

Figure 10.8 shows an example of violation, where `/var/cache /man/index.db` is modified to low integrity file during the installation of `2vcard`. SwInst identifies the chain of processes that have resulted in the violation. SwInst produces this invocation chain by tracing processes across clone, exec, and file open. During the commit phase, SwInst can then identify the dependency based on clone, exec and file open to help policy developers to generate policy. In this example, `mandb`'s post installation script was triggered because

`2vcard` created new mandb files that `mandb` need to process. SwInst resolves this conflict by marking `mandb` as trusted program and can read untrusted files specified in Figure 10.3.

There are two ways to resolve violations in SwInst. First is to ensure the content of the file is safe. SwInst supports using scripts (e.g., sh or awk) to validate files. The second method is to rely on identifying trusted programs in the installation. To facilitate the policy development process, SwInst tracks each process running during the installation. When files cannot be committed automatically, SwInst prints the process invocation chain leading to the modification of the problematic file. Users can then decide whether to write a validation script for the file, or designate some programs as trusted by creating new invocation rules. Since SwInst protects trusted processes against reading untrusted files, the result also lists out if there are any read violation occurred for trusted processes.

10.5.1 Installation-time policy

SwInst enforces different policies when installers are running. There are four entities: untrusted processes, trusted processes, `rudo` and the root helper.

Untrusted processes Untrusted processes run as *untrustedRoot*. SwInst allows untrusted processes to perform any operation within the confined environment, provided that these actions do not compromise the integrity of the trusted processes in the confined environment. Specifically, SwInst allows untrusted processes to:

- Read from any root-readable file
- Write to any root-writable file, except those that can compromise the integrity of trusted processes
- Connect to the root helper via IPC
- Transition to root when executing predefined trusted programs with predefined parameters

Trusted processes SwInst does not allow trusted processes to read anything untrusted. The integrity of trusted processes are protected using Spif by considering *untrustedRoot* as an untrusted user. Trusted processes also create an authorized deletion entry when deleting files.

`rudo` Similar to `uudo` in SPIF, `rudo` acts as a gateway for transitioning from untrusted processes to benign processes. `rudo` allows transitioning from untrusted processes into root only when specific conditions are met. Otherwise, the installation will fail.

Root helper SWINST uses root helper to provide both read and write accesses to untrusted processes. Unlike SPIF's helper process, the root helper in SWINST opens files inside COW as writable for untrusted processes; in addition, the root helper also marks the files as untrusted. Apart from opening files, the root helper also perform other file system operations such as chmod, unlink, chown, symlink, etc.. While untrusted processes can modify root files via the root helper, the root helper will only grant permissions to modify files that do not directly compromise the integrity of trusted processes within the confined environment. For example, attempts to replace the loader, system libraries, or SWINST-library will be denied.

10.5.2 Commit-time policy

File-based policy The simplest policy that SWINST supports is append-only policy. SWINST applies this policy mainly on log files such as `/var/log/apt/history.log` or `/var/log/apt/term.log` to make sure that new contents are only inserted at the end of the files. Some files such as `/var/lib/dpkg/available-old` maintain information about installed packages. SWINST ensures only new entries corresponding to the just installed packages are added to the files. SWINST uses file-based policy because `dpkg` modifies these files. Since `dpkg` will execute untrusted scripts, it is not safe to designate `dpkg` as trusted.

Invocation-based policy Figure 10.4 shows a list of trusted programs in SWINST. SWINST ensures the execution environment of these programs, and only allow them to consume untrusted files specified in certain directories. Files modified by these programs are therefore not checked.

Figure 10.5 also shows a list of programs that SWINST trusts to execute as benign process only when invoked with certain parameters.

```
/usr/bin/mandb
/usr/bin/fc-cache
/usr/bin/update-desktop-database
/usr/sbin/update-fonts-scale
/usr/sbin/update-fonts-dir
/usr/sbin/update-mime
/usr/bin/gtk-update-icon-cache
/usr/share/gnome-menus/update-gnome-menus-cache
/var/lib/dpkg/info/python-gmenu.postinst
/usr/sbin/update-alternatives
/usr/bin/update-alternatives
/usr/bin/gconftool-2
/usr/sbin/gconf-schemas
/usr/lib/libgtk2.0-0/gtk-update-icon-cache
/usr/bin/defoma
/usr/bin/mkfontscale
/usr/sbin/update-info-dir
/var/lib/dpkg/info/ureadahead.postinst
/var/lib/dpkg/info/doc-base.postinst
/usr/bin/update-mime-database.real
/usr/bin/update-gconf-defaults
/usr/sbin/update-xmlcatalog
/usr/bin/dpkg-divert
```

Figure 10.4: Trusted programs

```
/usr/sbin/useradd
/usr/sbin/groupadd
/usr/bin/chage
/usr/bin/dpkg-statoverride
/usr/sbin/usermod
/usr/sbin/userdel
```

Figure 10.5: Trusted programs with rules for parameter validation

Component	C/C++	Header	Other
shared	699	47	
library	536	42	47
wrapper	945		758
helper	1209	88	5
rudo	39		22
Total	3428	177	832

Figure 10.6: Code complexity of SwInst

10.6 Evaluation

We implemented SwInst on Ubuntu 10.04 based on Spif. SwInst extended Spif by providing library functions specific to installation within the isolated environment. SwInst also introduced root helper daemon for the installation and apt-get wrapper for creating isolated environment. Table 10.6 shows the complexity of SwInst in addition to Spif. SwInst only reuses some of the code from the Spif library.

To evaluate if SwInst is compatible with existing packages. We assigned packages from the universe and multiverse repositories as untrusted and installed them randomly. Each untrusted package may depend on other packages. SwInst installed the benign packages, and then the untrusted packages.

We tested 20540 unique packages out of the 23433 untrusted packages A total of 17863 installations were performed. 702 installations (4%) failed to install automatically because the installation requires user interaction, benign packages not installed successfully, or other implementation issues that our installer have not handle (e.g., handling DBus messages). 14964 installations (83.7%) were completed successfully without triggering any violation. The remaining 2197 installations (12.3%) were failed because of some violations/errors as listed in Figure 10.7

Out of the 1288 failed installations that invoked programs that untrusted installers should not invoke, over 1184 involve invoking `update-rc.d`. 77 involve `dkms`, 50 involve `depmod`, 29 involve `update-initramfs`, 8 involve `update-grub`. Some installations invoked multiple of these programs.

Count	Violation
18	Making an untrusted file setuid-to-root
48	Attempting to restart existing application
54	Package appears in both benign and untrusted packages
59	Involve package uninstallation which is not implemented
85	apt-get failed to retrieve packages
173	Package in neither benign nor untrusted repository
472	Benign package depends on untrusted packages
1288	Invoking programs that untrusted installer should not invoke

Figure 10.7: Number of installations failed to install due to violations

```
Write low: /var/cache/man/index.db 31382_2 /usr/bin/mandb
[31300_0] L /lwip/executables/dpkg/dpkg_original
[31373_0] L /lwip/executables/dpkg/dpkg_original
[31373_1] L /var/lib/dpkg/info/man-db.postinst  /var/lib/dpkg/info/man-db.postinst ...
[31373_2] L /bin/dash  /var/lib/dpkg/info/man-db.postinst triggered /usr/share/man
[31373_3] L /usr/share/debconf/frontend  /usr/share/debconf/frontend ...
[31373_4] L /usr/bin/perl  /usr/share/debconf/frontend /var/lib/dpkg/info/man-db.postinst ...
[31381_0] L /usr/bin/perl  /usr/share/debconf/frontend /var/lib/dpkg/info/man-db.postinst ...
[31381_1] L /var/lib/dpkg/info/man-db.postinst  /var/lib/dpkg/info/man-db.postinst ...
[31381_2] L /bin/dash  /var/lib/dpkg/info/man-db.postinst triggered /usr/share/man
[31382_0] L /bin/dash  /var/lib/dpkg/info/man-db.postinst triggered /usr/share/man
[31382_1]  L /usr/bin/perl  perl -e @pwd = getpwnam("man"); $( = $) = $pwd[3]; $< = $> = $pwd[2];
[31382_2] L /usr/bin/mandb  /usr/bin/mandb -pq
+     /var/cache/man/index.db
```

Figure 10.8: Invocation chain for installing `2vcard`
Invocation chain explaining why `/var/cache/man/index.db` was downgraded to low integrity during installation of `2vcard`

Chapter 11

Related work

In this chapter, we discuss a number of works related to the dissertation.

11.1 Malware detection and avoidance

The most widely adopted malware defense techniques are based on detection and avoidance. They attempt to detect and stop malware from running in the first place. Before any new piece of code and data can be used, these techniques attempt to determine if the file is free of malware. Anti-virus, Windows Security Zone [Microsoft, 2015a], and Mac OS X Gatekeeper [Apple Inc., 2015a] belong to this category. They all work by either blacklisting malware or white-listing files obtained from identifiable and verifiable sources. For instance, anti-virus relies on malware signatures. Windows Security Zone relies on the domains that the files come from. Gatekeeper uses code-signing with keys signed by Apple. We discuss each of them in detail below.

11.1.1 Anti-virus

Modern anti-virus software relies on pattern scanning. The idea is to first identify a set of characteristics that malware possesses, called patterns. The anti-virus software running on a client computer will then match every file with these patterns. A match would suggest that the file could be malware. As benign files may be marked as malware due to false positives, anti-virus software also uses white-listing. Upon detection, anti-virus software will proceed with remediation procedures such as prompting for user actions and quarantining or removing the suspicious files.

The success of pattern-based solutions depends on both the expressiveness and coverage of the patterns for identifying malware. The simplest form of patterns uses hashing, e.g., cryptographic hashing functions MD5 or SHA, which generates a unique checksum for each unique file.

Cryptographic hashing functions have an avalanche effect: a single bit-flip in the file would result in a completely different pattern. Malware can therefore easily evade detection using techniques such as polymorphism or appending random data. When two files contain mostly identical contents, the fact that one file is malware suggests that the other file is also likely to be a malware. The anti-virus industry therefore introduced context triggered piecewise hashes (CTPH, a.k.a. Fuzzy hashes), e.g., ssdeep [Kornblum, 2006]. These hashes can match inputs that have homologies. Files with mostly the same but slightly different content will yield hash values with common substrings. Malware that shares some code could therefore be captured using CTPH.

These techniques, however, may not detect metamorphic malware, which has different code yet with the same semantics. A common technique to solve the problem is to generate byte-patterns rather than relying on file hashes. Yara [Alvarez, 2015] is a popular pattern matching tool for byte sequence matching in malware. Instead of using a summary Byte-pattern captures the essence of malicious behaviors by identifying the corresponding instructions. As malware can be encrypted, bytes can be obfuscated. Anti-virus vendors therefore are also analyzing process memory during run-time to scan for patterns. Volatility [The Volatility Foundation, 2015], a memory dump tool on Windows, is often combined with Yara to identify malware.

Apart from analyzing malware statically, anti-virus vendors also use dynamic analysis to identify runtime malicious behaviors (e.g., cuckoo [Cuckoo Foundation, 2015], a platform for automatically testing malware). During program executions, system calls or Windows API calls are monitored and matched against known malicious or suspicious behaviors. These patterns are usually defined manually, e.g., popular malware Flame [Kaspersky Lab] creates mutex of names in the form of `__fajb.*` or `DVAAccessGuard.*`, Turlacomrat [Tanase, 2015] moves files with names `Microsoft\shdocvw.tlb`, `Microsoft\oleaut32.dll`, ... Other patterns include creating remote thread in other processes, installing and communicating via tor network, detecting virtualized environment, or installing OpenCL library (for mining Bitcoins) [Cuckoo Foundation, 2015].

Pattern-based approaches are effective only when anti-virus vendors have access to the malware before their clients so that vendors can generate patterns for clients to detect and block the malware. Before 2000s, the total unique malware samples were less than 100,000; however, the number of new malware found in the year 2014 alone already reached 148,000,000 [McAfee Labs, 2015], which is more than 4.5 malware creations per second. The rate of new unique malware becomes so high that pattern-based approaches are no longer effective because (1) anti-virus vendors may not have seen the malware before, and (2) the pattern may not be delivered to the clients in time; furthermore, malware started to employ different techniques to detect virtualized environment, which is commonly used by anti-virus vendors to analyze malware but not typical among end users. By exhibiting legitimate behaviors during analysis, malware can evade detection.

11.1.2 Origin-based protection

Anti-virus software relies on databases of known malware and good-ware. Every piece of data on the system is checked against known malware patterns. Anti-virus therefore cannot protect against new malware that has not be seen by anti-virus vendors. Instead of relying on anti-virus vendors, Windows Security Zone relies on the origins of the data: user's trust on a file depends on where the file comes from. For example, files coming from the OS distributor or local network would be more trustworthy than files coming from the Internet.

Windows Security Zone maps domains into zones of different trustworthiness. Windows predefined five zones: URLZONE_LOCAL_MACHINE, URLZONE_INTERNET, URLZONE_TRUSTED, URLZONE_INTERNET, and URLZONE_UNTRUSTED. These zones correspond to different security boundaries that users commonly have. Users can also define additional zones. When files are downloaded from the Internet, applications can fill in the zone information by calling some system APIs. Windows stores zone information along with files using as Alternate Data Stream on NTFS, similar to extended attributes on EXT file systems. The zone information is not used by the OS, and the OS enforces no policy based on it. Processes running executables from the Internet do not have special labels. It is up to applications to decide how to use the zone information when consuming the files. For instance, Windows Explorer will prompt users when attempting to execute files from the zone URLZONE_INTERNET or URLZONE_UNTRUSTED.

Microsoft Office will run in Protected View [Microsoft, 2015b], a mode that makes Microsoft Office harder to be exploited at the cost of reduced functionality, when consuming files with these labels too. This limits damages that a compromising Microsoft Office can inflict.

OS X does not provide fine granularity classification of file origins as in Windows Security Zone. Gatekeeper [Apple Inc., 2015a] in OS X functions similarly to Windows Security Zone, and it will prompt users for confirmation when running Internet executables unless the integrity of the executables can be verified (Section 11.1.3). Gatekeeper stores the origin information as extension attribute [Lin, 2013] along with the file. Note that both Security Zone and Gatekeeper rely on applications that perform downloading to label files properly by invoking some APIs. These APIs will then fill in the corresponding information. In OS X, a flag is set to label Internet files. Files that do not have such information will be treated as regular user files and will not trigger any prompting.

11.1.3 Code-signing

Apple (OS X and iOS) relies heavily on code signing to identify and verify the origin of the code. The technique itself does not protect against malicious code, but simply a way to provide a secure end-to-end channel to distribute code from code producers to code consumers.

Code signing imposes no real restriction on malware writers. Malware writers can still get a key to distribute signed malware [F-Secure Labs, 2013] or invoking private APIs [Wang et al., 2013]. The only restriction that code signing imposes is via the code review process. For apps distributed through App Store, Apple relies on manual code review to identify malicious applications. Each app is reviewed to ensure that every permission the app needs has legitimate reasons. This is a lengthy and subjective process. Apps that violated Apple's policy or failed to demonstrate the need for the requested permissions will need to be modified. Apps that refuse to comply will be banned from the App Store, which is the sole mean of distribution for iOS devices. For OS X apps that are distributed outside App Store, Apple relies on verifying the identities of the developers. By default, GateKeeper only allows downloads from Mac App Store and identified developers. Apple has the ability to revoke a certificate when malicious activity is detected [Kim, 2011].

Apples model is built on trust rather than technical foundations. Manual code review is unreliable, and attacks can happen once the developers turned malicious. There have been incidents where malicious apps got published [Wang et al., 2013, Xing et al., 2015] because the app review process failed to identify malicious behaviors, or malware bypassed the Gatekeeper because an identified developer key was used maliciously.

Conclusion Since detection-based techniques have to block malware before they run, they are obstructive to functionality. As there is no way to be confident if a flagged item is actually a malware, users are often prompted to make the ultimate decisions as to whether to proceed with the execution. In the recent XcodeGhost [Gregg Keizer, 2015] incident, some of the Chinese iOS developers ignored warnings from Gatekeeper and used compromised versions of XCode to create backdoored iOS apps. These iOS apps were distributed via Apple App Store to users. This signifies the weaknesses of detection-based techniques.

11.2 Policy-based confinement

Recognizing the fact that it is impossible to fully characterize what a malware is, proactive approaches assume malware can run and aim at restraining what malware can do.

A natural (and perhaps the best studied) proactive defense is to sandbox potentially malicious code using policies based on the principle of least privilege. This approach can be applied to software from *untrusted* sources [Goldberg et al., 1996], which may be malicious to begin with; or to software from *trusted* sources [Loscocco and Smalley, 2001a, Ubuntu, 2015, Provos, 2003] that is benign to start with, but may turn malicious due to an exploit. In policy-based confinement, a reference monitor will check every operation that the code performs. The reference-monitor then decides whether to allow or block the operation when it is deemed as malicious.

The goal of policy-based confinement is to guard against improper use of resources. The most common form of resource to guard is the invocation of system-calls. Earlier OSes did not support policy enforcement at the system-call level, and research works had been focusing on developing supporting architectures (e.g., inside kernel space using kernel modules or in user-land such as ptrace [Padala, 2002] or delegation architecture [Garfinkel et al., 2004]). Linux introduced seccomp [Linux Kernel Organization, 2015] in 2005, a rule-based system-call filtering mechanism. seccomp itself is very limited because it does not allow policies beyond limiting a process to resources already granted. Other mechanisms such as LSM [Wright et al., 2002], TrustedBSD [Watson et al., 2003], Windows Integrity Mechanism [Microsoft, 2015c], and System Integrity Protection [Apple Inc., 2015b] (on OS X) focus primarily on security-sensitive operations. Both LSM and TrustedBSD implement hooks on security-sensitive operations to enforce policies on kernel objects (e.g., inodes and process structs). Windows Integrity Mechanism (WIM) attaches integrity labels to subjects and objects and enforces a policy that does not allow subjects with lower integrity labels to modify objects with higher integrity labels. System Integrity Protection protects Apple signed files and processes from being tempered by any non-signed process, including root processes. Apart from enforcing policies at the OS level, policies can also be enforced at the binary-level (e.g., SFI [Wahbe et al., 1993]) or below the OS level (e.g., Library OS [Porter et al., 2011, Tsai et al., 2014] or hypervisor [Butt et al., 2012].

11.2.1 Drawbacks for policy-based confinement

While the goal of policy-based confinement against malicious code is simple, there are several challenges for applying policy-based confinement to defend against malicious code:

Difficulty of policy development: Experiences with SELinux [Loscocco and Smalley, 2001a] and other projects [Acharya et al., 2000, Sekar et al., 2003, Ubuntu, 2015] show that policy development requires a great deal of expertise and effort. Policies depend highly on the usage environment and usage behavior. A slightly different configuration or an unanticipated usage behavior could result in policy violations. For example, Ubuntu has developed an AppArmor profile for Firefox; however, it is not enabled by default [Dziel, 2014, Mozai, 2013] due to false positives. The use of SELinux often deter system administrators from securing their own systems (e.g., by placing configuration files at a location other than the default location) because of the difficulties in reconfiguring SELinux policies.

Policies that provide even the modest protection from untrusted code can break benign applications. On the other hand, developing secure policies to protect against malicious code is also difficult. Malicious code can mimic benign code [Parampalli et al., 2008]. A vulnerability on OS X suggested the difficulty in developing policy especially when apps can interact— applications can store and share user credentials (e.g., browser logins) using KeyChain. Although applications can define their own access control lists to restrict what applications can access their KeyChain entries, OS X does not prevent other applications from deleting and recreating the entries to grant themselves access. This vulnerability has allowed attackers to gain access to user credentials since most applications do not check ACL permissions [Xing et al., 2015].

Subversion attacks on benign software: Even highly restrictive policies can be inadequate, as malware can co-opt benign applications to carry out prohibited operations: malware may trick a user to run a benign application in insecure ways or exploit vulnerabilities in benign applications to perform arbitrary actions, e.g., use a copy utility to overwrite a system library with malware. Alternatively, malware may exploit vulnerabilities in a benign application, e.g., create a malicious file on the desktop with an enticing name. When clicked on by the user, it compromises a benign application that opens the file, as in the Acrobat sandbox escaping vulnerability [Fisher, 2014]. Since this benign application is not confined by a sandbox, it can now perform arbitrary actions.

Difficulty of secure policy enforcement: Uncircumventable policies are usually enforced in OS kernels. The drawbacks of kernel-based approaches have been eloquently argued [Jain and Sekar, 2000, Garfinkel et al., 2004]: kernel programming is more difficult, leads to less portable code, and creates deployment challenges. Experience with various commercial containment mechanisms such as sandboxie [Sandboxie Holdings, LLC., 2015], Bufferzone [BufferZone Security Ltd., 2015], and Dell Protected Workspace [Dell, 2015] have demonstrated the challenges of building effective new containment mechanisms for malicious code [Rahul Kashyap, 2013]. Approaches such as ptrace [Padala, 2002] avoid these drawbacks by enabling policy enforcement to be performed in a user-level monitoring process; however, it poses performance problems due to the frequent context switches between the monitored and monitoring processes. Moreover, the monitoring process needs to protect itself against attacks launched from the confined processes.

More importantly, TOCTTOU attacks are difficult to prevent [Garfinkel, 2003]. Ostia [Garfinkel et al., 2004] avoided most of these drawbacks by developing a *delegating architecture* for system-call interposition. It used a small kernel module that permits a subset of "safe" system calls (such as `read` and `write`) for monitored processes, and forwards the remaining calls to a user-level process. Applications such as Chrome, Adobe Reader and Internet Explorer adopted similar model (See Section 11.2.2).

Some works such as SELinux [Loscocco and Smalley, 2001b], Systrace [Provos, 2003] and AppArmor [Ubuntu, 2015] focus on protecting benign code, and they typically rely on training to create a policy. Such a training-based approach is inappropriate for untrusted code. So Mapbox [Acharya et al., 2000] develops policies based on expected functionality by dividing applications into various classes. Model-carrying code [Sekar et al., 2003] proposes a framework in which code producers and code consumers can effectively collaborate to come up with policies that give applications sufficient privileges to function. While it represents a significant advance over purely manual development of policies, it still does not scale to large numbers of applications. Rather than confining arbitrary operations, WIM and System Integrity Protection aim to protect the system itself by preventing untrusted processes from modifying the system. However, WIM and System Integrity Protection do not impose any limitations on system processes. They can still get compromised when consuming untrusted data.

Instead of developing policies to protect against arbitrary untrusted code proactively, policy-based confinement is more commonly used as a mechanism to deter attackers from exploiting applications in the first place. OSes such as iOS, Android, and app models in Windows and OS X predefine a set of permissions. Application developers declare what permissions their applications need. OSes grant only the requested permissions to the app regardless of whether the app is compromised or not. It becomes less attractive for malware writers to compromise apps as they cannot gain as many privileges as compromising an unconfined application. Clearly, this approach cannot protect malware that was distributed as apps because malware writers can simply declare whatever permissions they need and abuse them; furthermore, the permission systems in the OSes are getting more complicated over time— Android API level 3 has 103 permissions, and has increased to 165 permissions in API level 15 [Wei et al., 2012]. iOS and OS X also added more entitlements over time. It is not surprising that some of the Apple applications in OS X are sandboxed, yet they are granted with special entitlements that allow them to circumvent some of the restrictions [letiemble, 2011].

11.2.2 Privilege separation

Privilege separation techniques extend policy-based confinement approaches to support applications that require significant access to realize their functionality. The application is decomposed into a small, trustworthy component that retains significant access and a second larger (and less-trusted) component whose access is limited to that of communicating with the first component in order to request security-sensitive operations. While policy-based confinement can confine malicious as well as frequently targeted benign applications (e.g., browsers), privilege separation is applied only to the latter class. Chromium browser [Reis and Gribble, 2009], Acrobat Reader and Internet Explorer are some of the prominent applications that employ privilege separation, more popularly known as the *broker architecture.* These applications isolate their renderers, which are complex and are exposed to untrusted content. Workers were given just enough privileges to work. As a result, vulnerabilities in a renderer (or more generally, a *worker*) process won't allow an attacker to obtain all privileges of the user running the application.

Privilege separation shifts the policy development responsibility to developers. Instead of having OS distributors or system administrators to configure policies, software developers already encode in policies what legitimate accesses the workers need. Since developers know exactly what accesses their programs need, they can develop good policies without compromising usability and functionality.

However, given the large effort needed to (a) develop policies and (b) modify applications to preserve compatibility, it is no wonder that in practice, confinement techniques are narrowly targeted at a small set of highly exposed applications. This naturally leads attackers to target sandbox escape attacks: if the attacker can deposit a file containing malicious code somewhere on the system and trick the user into running this file, then this code is likely to execute without confinement (because confinement is being applied to a small, predefined set of applications). Alternatively, the attacker may deposit a malicious data file, and lure the user to open it with a benign application that isn't sandboxed. In either case, the attacker is in control of an unconfined process that is free to carry out its malicious acts.

As a result of these factors, policy-based confinement can only shut out the obvious avenues, while leaving the door open for attacks based on evasion (e.g., Stuxnet [Falliere et al., 2011]), policy/enforcement vulnerabilities (e.g., sandbox escape attacks on Adobe Reader [Fisher, 2014], IE [Li, 2015] and Chrome [Constantin, 2013]), or social engineering. Stuxnet [Falliere et al., 2011] is a prime example here: one of its attacks lures users to plug in a malicious USB drive into their computers. The drive then exploits a link vulnerability in Windows Explorer, which causes it to resolve a crafted *lnk* file to load and execute attacker-controlled code in a DLL.

11.3 Isolation-based approach

An alternative to policy-based confinement is isolated-execution of untrusted code. The main advantage of isolation-based approach is its simple policy—isolation simply virtualizes all resources, and hence it does not have to decide whether an operation is allowed or denied. The underlying implementation can depend on policy-based approaches to deny access to shared resources. There are two types of isolation: One-way isolation and two-way isolation.

One-way isolation [Liang et al., 2003, Sun et al., 2005] permits untrusted software to read shared resources, but its outputs are held in isolation. This is usually used to isolate a less trustworthy security domain from a trustworthy security domain. One-way isolation is typically implemented with copy-on-write file systems. Commercial product Sandboxie [Sandboxie Holdings, LLC., 2015] realizes one-way isolation on Windows so that applications running inside the sandbox can modify any file without affecting the actual system. Bromium [Bromium] leverages virtualization technologies to create "micro-VMs" whenever users run an application. Two-way isolation protects integrity and confidentiality by limiting both reads and writes, holding the inputs as well as outputs of untrusted applications in an isolated environment. A classical example is to use air gap to physically separate different security-level networks. In cloud computing, virtual machines are widely used to provide isolation while allowing consolidating different security domains applications to run on the same physical machine. The app model on Android, iOS, OS X, and Windows 8 are based on this two-way isolation model. Apps cannot interact with each other by default.

Isolation approaches provide a stronger protection against malware since they block all interactions between untrusted and benign software, thereby preventing subversion attacks. Isolation approaches also provide much better usability because they permit sufficient access for most applications to work.

11.3.1 Drawbacks when applying on desktop environment

While the only policy that isolation-based approach enforces is to isolate resources, they too have several significant drawbacks, especially when applied in the desktop environment:

Fragmentation of user-data: Unlike policy-based confinement, which continues to support the model of a single namespace for all user-data and resources, isolation causes fragmentation: user-data and resources are partitioned into multiple containers, each representing a disjoint namespace. In app model, each app has its own home directory. App-created files are considered as app-data rather than user-data. In desktop environment such as Linux Container [Canonical Ltd., 2012], applications from one isolation context cannot be used in another context. Users therefore have to install and manage the same application across multiple contexts. Apiary [Potter and Nieh, 2010] proposed using unioning file system to simplify applications management.

Inability to compose applications: The hallmark of today's desktop OSes is the ability to compose applications. UNIX pipelines represented one of the early examples of application composition. Other common forms of composition can happen through files or scripts, e.g., saving a spreadsheet into a PDF file and then emailing this PDF file. Unfortunately, strict isolation prevents one application from interacting with any data (or code) of other applications, thus precluding composition.

No protection when isolation is breached: Strict isolation may be breached either due to a policy relaxation or through manual copying of files across isolation-contexts. Any malware present in such files can subsequently damage the system as these files do not carry any identifiers.

11.3.2 Desktop application

Isolation has been made popular by the app models. Despite of applying isolation on a completely new ecosystem, app models need to address the challenges above. The app models on iOS, Android, and Windows address the first two drawbacks by introducing new mechanisms for apps to interact. For instance, Android supports *intent* as an interaction mechanism– each app declares in its *intent filter* what resource type and actions the app is capable of handling. When an app needs to interact with another app, it creates an intent, which is an IPC mechanism provided by Android. Android will resolve the intent by picking an app which is capable of handling the intent request. Upon resolution, Android will transfer the control to the selected app. Intents can request data in multiple forms. For example, an app invoking a camera app for taking a picture can get the raw bitmap of an image directly, save the image to the app's selected private location, or ask the camera app to save the image to a public location and return the location. This model is much safer than the desktop model because all interactions are made explicit: apps only access data that they can handle, and they only share data that they are willing to share. By default, apps accept no intent. On the other hand, if an app does not support sharing, users have no way to access the data from other apps. Since apps need to use the new sharing mechanisms, desktop applications cannot simply run as apps; in addition, isolation environment has imposed a lot of restrictions on accessing system resources. As such, most of the desktop applications (e.g., Microsoft Office, Adobe Reader, Photoshop) do not support all functionalities when running as apps.

Instead of defining a completely new sharing mechanism, app model in OS X aims at reassembling the user-familiar unified file system view while enforcing file system isolation. OS X has applied *App Sandbox* on most system processes. It also mandates all apps distributed via App Store to run inside a sandbox. OS X developers spent tremendous efforts in preserving the normal desktop experience for isolated apps. OS X introduced PowerBox [Apple Inc., 2014] to grant apps access to user-files based on user interactions. This solves the fragmentation problem. When users need to open files, the apps will make IPC requests to a trusted daemon process running outside of the sandbox. The daemon process will then draw a file selection dialog box on behalf of the isolated app. Once users selected the file, the daemon will generate a token for the sandboxed app. The sandboxed app can then present the token to the kernel. The kernel will then permit the app to access the file.

While the mechanism is simple, OS X developers have spent a lot of efforts in ensuring the looks-and-feels of the dialog box matches with application styles. To make the technology usable in different scenarios, OS X extended PowerBox with *security-scoped bookmarks*. For example, users can configure web browser to download files to a user-specified folder. It is inconvenient if users have to select the same location via the file dialog box every time to grant the web browser permission to create files there. App-scoped bookmark allows apps to gain persistent accesses to a previously selected location. Apps are free to store the tokens generated by the trusted daemon for later use. OS X also introduced document-scoped bookmarks to solve another usability problem. Document-scoped bookmarks are tied to files— when users grant an app to access a file, the app can automatically access another file. This is useful in scenarios where multiple related files need to be accessed simultaneously. For example, a movie file can have a bookmark to a subtitle file. A html file can have bookmarks to all the embedded objects. This would allow a movie player or web browser to display the content properly. PowerBox does not solve the problem of composing applications. Indeed, OS X does not sandbox shell scripts invoked by sandboxed processes as long as the scripts were placed at a specific location outside of the sandbox. Some of the popular apps such as Adobe Acrobat and Photoshop do not run within App Sandbox. Developing a usable isolation environment for desktop OSes remains to be a hard problem [Reddit Discussion, 2014].

Microsoft Office introduced Protected View [Microsoft, 2015b] to confine itself when consuming untrusted data. The idea is to run the application in an one-way isolation environment if the application could be compromised by consuming the untrusted data. This effectively isolate the effect of possible exploitation. Protected view leverages WIM and application-awareness to achieve isolation. When consuming untrusted files, Microsoft Office will run itself as a low-integrity process. By default, system files and registry entries are of high-integrity, and user files and registry entries are of medium integrity. WIM prevent lower-integrity processes from writing into higher-integrity objects. The office process can therefore read but not write into system or user files. Since the office process itself needs to modify some files (e.g., temporary files), the process will write into some OS-designated low-integrity areas instead of the regular medium-integrity locations. Protected View still requires applications to be aware of being isolated so that they won't modify files that are of higher integrity.

Using userid as an isolation mechanism has been demonstrated in systems like Android and Plash [Seaborn, 2015] for isolating applications. One of our contributions is to develop a more general design that not only enforces strict isolation between applications, but also permits controlled interactions. Although Android also allows applications to interact, such interactions can compromise security, becoming a mechanism for a malicious application to compromise another high-integrity application. In contrast, SPIF ensures that malicious applications cannot compromise high-integrity processes. Furthermore, SPIF requires no modifications to applications, whereas Android requires applications to be rewritten so that applications do not violate the strict isolation policy.

Both SPIF and Plash [Seaborn, 2015] confine untrusted programs by executing them with a user id that has limited accesses in the system. Both SPIF and Plash use a helper process to grant additional accesses to confined processes. However, SPIF's focus is on preserving compatibility with a wide range of software, while giving concrete assurances about integrity and availability. SPIF achieves this goal by sandboxing all code, whereas Plash focuses on sandboxing only untrusted code with least privilege policies.

Another difference between our system and Android is that the Android model introduces a new user for each application, whereas we introduce a new (untrusted) user for each existing user.

11.4 Information flow control

Isolation separates resources into different isolation contexts based on security domains. While isolation is effective in protecting one domain from another, it also limits the ability to compose applications. To allow sharing and app composition, various isolation-based approaches introduced their own sharing mechanisms to circumvent isolation. Once sharing happens across an isolation boundary, isolation can no longer provide any protection as isolation does not provide finer-granularity tracking within a domain.

A natural extension to isolation is to attach labels to every subject and object and enforce policies to confine their interactions. Bell-LaPadula is one of the earliest multi-level security models concerning confidentiality. Its labels, ranked from highest confidential to least confidential, are top secret, secret, confidential, and unclassified. Subjects with clearance C can read information of label C or below (no-read-up); similarly, any output from the subjects can contain information derived from C, and hence is labeled as

C (no-write-down). Biba [Biba, 1977] focuses on integrity with labels from highest integrity to lowest integrity. It enforces no-read-down and no-write-up policies.

11.4.1 Usability problems with IFC policies

There are two classical integrity policies: Biba and low-water-mark [Biba, 1977]. They have been proposed for more than 40 years. A policy is nothing but defining what subjects can and cannot do. Operations that are not allowed by the policy will be denied. Naturally, if a policy does not deny any action that would have been succeeded on an unprotected system, user experiences would be preserved. This could lead to better usability. Since different policies allow different set of operations, therefore different policies have different usability.

Biba model enforces a strict separation between high and low-integrity objects and subjects, which impacts its usability. Consider a utility application such as a word-processor that needs to operate on both high and low integrity files. It would be necessary to have two versions of every such application, one for operating on high-integrity files and another for low-integrity files. It is cumbersome to install and maintain two versions of every application. Worse, a user needs to be careful in selecting the correct version of an application for each task — choosing a high-integrity version of an application for processing low-integrity files (or vice-versa) will lead to security failures and/or application crashes.

The low-water-mark policy avoids these drawbacks of the strict policy by permitting subject integrity to be downgraded at runtime. In particular, low-water-mark allows applications to be invoked with high integrity, and the integrity level can be downgraded if the application subsequently reads a low integrity object. Any operations allowed in Biba would also be allowed by the low-water-mark policy. Intuitively, low-water-mark has better usability. Fraser [Fraser, 2000] argues eloquently why low- water-mark policy has significantly better compatibility with existing software as compared to the strict model. However, prior to his LOMAC project, the low-water-mark policy was not very popular because of the *self-revocation* problem [Fraser, 2000]. Specifically, consider a subject that has already opened a high integrity file for writing. If this subject subsequently opens a low integrity file for reading, it is downgraded. At this point, the subject cannot be permitted to write into the high integrity file any more. Applications usually handle security failures when opening files, but once opened, they assume that subsequent read and write operations will not fail. When this assumption is invalidated, applications may malfunction or crash.

On one hand, the low-water-mark policy seems more usable because it allows more actions that would have be succeeded on an unprotected system. On the other hand, it suffers from self-revocation, which breaks usability.

11.4.2 Modern application of IFC

Usability remains one of the major concern in applying information flow tracking. The most notable widely deployed system is WIM. While WIM is more commonly used as an isolation mechanism, WIM is an instance of information flow tracking. Microsoft introduced WIM in Windows Vista to protect system objects against malicious modification. Most Windows users themselves are system administrators and login to Windows with their administrator accounts. As users run both regular applications and administrative applications with the same user identifiers, Windows XP has no way to differentiate if an administrative operation is initiated by users or malicious applications. Windows uses WIM to encode the "trustworthiness" of the processes. When administrative users login, Windows Explorer will have two security tokens, one as normal user and one as administrative user. These security tokens are for authorizing user actions. Whenever users run system applications or installers, Windows Explorer will prompt users for confirmation. Only then the programs will run with administrative tokens (high integrity-level) and be able to write into high-integrity system objects; otherwise, programs will run with normal user tokens, i.e., medium integrity-level and can only modify user files. WIM ensures that medium integrity-level processes spawn only medium integrity-level processes. While WIM enforces information flow policy, this policy is not sufficient to protect system integrity. WIM enforces only no-write-up to protect system objects, but it does not enforce no-read-down. This is because enforcing no-read-down would break system applications and result in self-revocation. As a result, attacks have been successfully carried out— an attack vector of Stuxnet modifies a user-writable task XML file maliciously so that an attacker-specified program will run with the system privilege [jduck, 2014].

Most of the recent information flow tracking techniques remain in the research areas. LOMAC focuses on addressing self-revocation. It extends low-water-mark to address some of the common self-revocation scenarios in pipes using heuristics. It does not consider the direction of the pipes and hence could stop a safe execution that does not have self-revocation. Furthermore, LOMAC does not address problems for processes connected via other means such as shared memory or sockets.

PPI [Sun et al., 2008b], UMIP [Li et al., 2007] and IFEDAC [Mao et al., 2011] focus on applying information flow tracking on commodity OSes, but they all require significant changes to the OS kernel and only work for open source OSes. UMIP [Li et al., 2007] focuses on protecting system integrity against network attackers. UMIP uses the sticky bit to label low-integrity data. The kernel tracks process integrity levels and enforce policies on low integrity processes. High integrity processes remain high integrity as long as they do not consume any low-integrity files or interact with low-integrity processes. Low integrity processes can only read from non-user files and write into world-writable files. This is true for network applications but not desktop applications. UMIP is designed to protect servers. Compromising user files and processes is an important avenue for malware propagation, but UMIP does not attempt to protect the integrity of user files. Downgrading a user process allows no interactions with the user files, making the system not usable for desktops.

IFEDAC [Mao et al., 2011] extends UMIP to protect against untrusted users as well by tracking not only network provenance information, but also local user. This is the same as our system. Instead of labeling files using sticky bit as in UMIP, IFEDAC uses kernel to track object labels as well. This allows IFEDAC to have arbitrary number of labels.

PPI [Sun et al., 2008b] is designed to preserve integrity by design and focuses on automating policies. PPI relies on exhaustive training to determine policies for subjects and objects. It starts by manually defining a set of integrity-critical objects. The set of subjects and objects that need to remain at high integrity is then expanded recursively based on the observations from training. The absence of training can lead to failures of high-integrity processes. Specifically, incomplete training can lead to a situation where a critical high-integrity process is unable to execute because some of its inputs have become low-integrity, leading to availability problem.

Spif and the majority of the Flume [Krohn et al., 2007] implementation locates in the user-space. Flume, like Ostia, requires a kernel module to confine processes by blocking some system calls. Flume prevents confined processes from performing fork because of the complexity in maintaining labels for all opened resources. As a result, Flume allows only spawn (fork

+ exec) but not fork. Furthermore, child processes cannot inherit any file descriptors except pipes from parent processes. This constrains the applications that Flume can run. Spif relies on existing userid mechanism for both labeling and confining processes. Spif does not constraint the set of system calls that processes can make – they can fork and inherit file descriptors as usual.

While both Spif and UMIP rely permission bits to encode integrity labels, Spif also uses the mechanism to enforce policy. Furthermore, UMIP can only encode one bit information. Spif relies on userid, hence can encode richer provenance information.

PPI [Sun et al., 2008b] is designed to preserve integrity by design and focuses on automating policies. But there are several important advances we make in Spif over PPI. First, Spif uses a portable implementation that has no kernel component, whereas the bulk of PPI resides in the kernel. Second, PPI approach for policy inference requires exhaustive training, the absence of which can lead to failures of high-integrity processes. Specifically, incomplete training can lead to a situation where a critical high-integrity process is unable to execute because some of its inputs have become low-integrity. The approach presented in this work avoids this problem by preventing any high-integrity file from being overwritten with low-integrity content. On the other hand, PPI has some features that we don't: the ability to run low-integrity applications with root privilege and dynamic context switch from high to low integrity. Spif does not have these features because these features significantly complicate the system design and implementation.

On the other hand, PPI has some features that Spif does not have: the ability to run low-integrity applications with root privilege and dynamic context switch from high to low integrity. Spif does not have these features because these features significantly complicate the system design and implementation. Moreover, Spif avoids this problem by preventing any high-integrity file from being overwritten with low-integrity content, and hence can preserve the availability of high integrity processes.

11.4.3 Decentralized information flow control

Traditional IFC systems have labels predefined by system administrators. They focus on protecting system integrity or preventing confidential data leakage. Labels in IFC systems have system-wide meanings. In contrast, DIFC [Myers and Liskov, 1997] relaxes the notion of labels. Applications can create their own labels and these labels are only meaningful to the application creators, and do not necessarily have meanings to the system and other applications. The DIFC system will propagate and enforce system-predefined policies based on the labels. This allows composition of different applications while satisfying the security requirements for the applications and data.

DIFC approaches can be classified into language-based and OS-based. Language-based solutions (e.g, Jif [Myers et al., 2001]) push information-flow control inside applications. Developers have to write their programs with new language primitives and design the program logic to comply with information-flow policy; otherwise, programs will not be protected or simply do not run due to violations. An advantage of language-based approach is that labels can be assigned and enforced at a fine-grained level such as variables.

OS-based solutions such as Flume [Krohn et al., 2007], HiStar [Zeldovich et al., 2006] and Asbestos [Efstathopoulos et al., 2005] enforce policies at the OS subjects and objects. They usually require OS changes to track information flow. The advantage of OS-based solutions is that they usually require less changes to applications. For example, HiStar redesigned the OS with new abstractions to allow more precise labeling at memory-page level. While the majority of the code in Flume resides in user-space, Flume still relies on kernel modules to confine processes. This makes Flume hard to support different Linux distributions and other Unix systems.

Laminar [Porter et al., 2014] is both a language- and OS-based solution. It requires minimal changes to existing application code. Instead of rewriting the entire application to comply with information-flow policies, application developers only need to indicate which parts of the code need access to sensitive data, and what capability a thread will have when executing the code. The JVM in Laminar will then perform the runtime checking and policy enforcement. Laminar can also use labels from OS objects such as files, making it a combination of both OS and language based solution.

Chapter 12

Conclusions and Future Work

Todays OSes adopt users as the basic unit of trust. Every file and process owned by the same user has the same userid as the user. This design stems from the very first multi-user OS created, a time when computers were self-contained, and file contents were under the control of users. Today, users frequently download data and code from the Internet, without fully understanding their content or consequences. However, existing desktop OSes reuse the same old trust model and treat downloaded files as if users are fully responsible for them. This trust is exploited by today's malware.

In this dissertation, we started with the hypothesis that incorporating remote origins information of the Internet-downloaded files can enhance system security for todays OSes. We tested the hypothesis by answering questions from three perspectives: (1) How to support provenance-tracking enforcement mechanisms on todays OSes? (2) How to develop usable provenance-based policies? (3) How useful and practical are provenance-tracking techniques when applied on todays OSes?

Like most existing approaches, we started with a kernel-based enforcement mechanism for labeling provenances and enforcing policies. We developed a mechanism based on LSM on Ubuntu Linux. This approach, however, highly dependent on the kernel and is error-prompt. We then started exploring the possibility of implementing enforcement mechanisms at the user-level. As OSes already track every resource and enforce policies using userid, we naturally explored the use of DAC permission. We proposed a novel dual-sandboxing architecture to provide system-wide provenance tracking and policy enforcement mechanism. By splitting the confinement of benign and untrusted processes, we can overload the existing DAC permission transparently for provenance-tracking. This has allowed us to develop the system on Ubuntu and PC-BSD, and we later ported the system to Windows as well.

Policies are as important as the enforcement mechanisms. We first focused on integrity-preservation policies. Information flow tracking-based integrity policies have been proposed for more than 40 years, yet there has been only a very limited adoption. This is because these policies change application behaviors and can severely affect the system usability. We proposed a model to formalize the usability versus functionality trade-off made by various integrity-preservation policies. We showed that existing policies do not provide an optimal trade-off. Towards this end, we proposed a couple of new policies. We formally prove that our proposed policies preserve the integrity and availability. We then generalized the integrity policy to a policy language that principals can express their security requirements. Instead of preserving integrity, the goal is to provide a general security policy framework that can model different policies. Our policy language can model and improve security of policies such as Bubbles, Android, and Web.

With enforcement mechanisms and policies in place, we built systems to understand how useful and practical these provenance-tracking systems can be. We first built an integrity-preserving system to defend against malware. The system incurs low performance overhead and is agnostic to both OSes and applications. It has been implemented on Linux, BSD, and Windows, supporting large, unmodified applications like Firefox, Microsoft Office, Adobe Reader and Photoshop. It can also protect systems against high-profile malware such as Stuxnet and Sandworm. The second system we built addresses a long-standing problem in information flow tracking, called self-revocation. We implemented the policy that has optimal trade-off between usability versus functionality on Ubuntu Linux. During the development of the policy, we realized that there exists another class of policy which can achieve both usability and functionality using rollback. We showed that there are situations where rollback is necessary to achieve functionality and usability. We illustrated this using a secure software installation system. We demonstrated the need for rollback and commit capabilities.

Throughout this dissertation, we have shown that tracking provenance information is both useful and practical on todays OSes. Indeed, by capturing provenance information, we can have a more generalized policy framework that can not only model popular security systems, but also improve their securities.

As part of the future work, we plan to further explore the relationship between rollback and existing security policies such as security automaton. We are interested in understanding if the policy expressiveness power of rollback mechanisms is strictly more powerful than security automaton. We are also interested in implementing the generalized policy on other existing platforms such as Android and web browsers.

Bibliography

[Acharya et al., 2000] Acharya, A., Raje, M., and Raje, A. (2000). MAPbox: Using Parameterized Behavior Classes to Confine Applications. In *USENIX Security*.

[Aho and Corasick, 1975] Aho, A. V. and Corasick, M. J. (1975). Efficient String Matching: An Aid to Bibliographic Search. In *Communications of the ACM 18(6)*.

[Alvarez, 2015] Alvarez, V. M. (2015). yara— the pattern matching swiss knife for malware researchers (and everyone else). https://plusvic.github.io/yara/. Online; accessed October 13, 2015.

[Apple Inc., 2014] Apple Inc. (2014). App sandbox design guide. https://developer.apple.com/library/mac/documentation/Security/Conceptual/AppSandboxDesignGuide/AppSandboxDesignGuide.pdf. Online; accessed November 12, 2015.

[Apple Inc., 2015a] Apple Inc. (2015a). OS X: About Gatekeeper. https://support.apple.com/en-us/HT202491. Online; accessed October 13, 2015.

[Apple Inc., 2015b] Apple Inc. (2015b). System integrity protection guide. https://developer.apple.com/library/prerelease/ios/documentation/Security/Conceptual/System_Integrity_Protection_Guide/System_Integrity_Protection_Guide.pdf. Online; accessed November 12, 2015.

[Biba, 1977] Biba, K. J. (1977). Integrity Considerations for Secure Computer Systems. In *Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts*.

[Bromium] Bromium. Understanding bromium micro-virtualization for security architects. Available from: http://www.bromium.com/sites/default/files/Bromium%20Microvirtualization%20for%20the%20Security%20Architect_0.pdf.

[Brumley and Song, 2004] Brumley, D. and Song, D. (2004). Privtrans: Automatically Partitioning Programs for Privilege Separation. In *USENIX Security*.

[BufferZone Security Ltd., 2015] BufferZone Security Ltd. (2015). BufferZone. http://bufferzonesecurity.com/. Online; accessed September 18, 2015.

[Butt et al., 2012] Butt, S., Lagar-Cavilla, H. A., Srivastava, A., and Ganapathy, V. (2012). Self-service cloud computing. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 253–264, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/2382196.2382226.

[Canonical Ltd., 2012] Canonical Ltd. (2012). lxc linux containers. http://lxc.sourceforge.net/. Online; accessed November 14, 2012.

[Claudio nex Guarnieri and Alessandro jekil Tanasi, 2015] Claudio nex Guarnieri and Alessandro jekil Tanasi (2015). Malwr. https://malwr.com.

[Constantin, 2013] Constantin, L. (2013). Researchers hack Internet Explorer 11 and Chrome at Mobile Pwn2Own. http://www.pcworld.com/article/2063560/researchers-hack-internet-explorer-11-and-chrome-at-mobile-pwn2own.html. Online; accessed September 18, 2015.

[Cuckoo Foundation, 2015] Cuckoo Foundation (2015). Automated malware analysis - cuckoo sandbox. http://www.cuckoosandbox.org/. Online; accessed October 13, 2015.

[Dell, 2015] Dell (2015). Dell Data Protection — Protected Workspace. http://www.dell.com/learn/us/en/04/videos~en/documents~data-protection-workspace.aspx. Online; accessed May 11, 2015.

[Dziel, 2014] Dziel, S. (2014). [Bug 1322738] [NEW] Apparmor prevents the crash reporter from working. https://lists.ubuntu.com/archives/ubuntu-mozillateam-bugs/2014-May/148059.html. Online; accessed April 8, 2016.

[Efstathopoulos et al., 2005] Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazières, D., Kaashoek, F., and Morris, R. (2005). Labels and Event Processes in the Asbestos Operating System. In *SOSP*.

[F-Secure Labs, 2013] F-Secure Labs (2013). Mac Spyware: OSX/KitM (Kumar in the Mac). https://www.f-secure.com/weblog/archives/00002558.html. Online; accessed November 12, 2015.

[Falliere et al., 2011] Falliere, N., Murchu, L., and Chien, E. (2011). W32. Stuxnet Dossier. *White paper, Symantec Corp., Security Response*.

[Fisher, 2014] Fisher, D. (2014). Sandbox Escape Bug in Adobe Reader Disclosed. http://threatpost.com/sandbox-escape-bug-in-adobe-reader-disclosed/109637. Online; accessed September 18, 2015.

[Fraser, 2000] Fraser, T. (2000). LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *S&P*.

[Garfinkel, 2003] Garfinkel, T. (2003). Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *NDSS*.

[Garfinkel et al., 2004] Garfinkel, T., Pfaff, B., and Rosenblum, M. (2004). Ostia: A Delegating Architecture for Secure System Call Interposition. In *NDSS*.

[Goldberg et al., 1996] Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. A. (1996). A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker). In *USENIX Security*.

[Google, 2016] Google (2016). android. https://www.android.com/. Online; accessed April 7, 2016.

[Google Security Research, 2014] Google Security Research (2014). Windows Acrobat Reader 11 Sandbox Escape in MoveFileEx IPC Hook.

https://code.google.com/p/google-security-research/issues/detail?id=103. Online; accessed September 18, 2015.

[Gregg Keizer, 2015] Gregg Keizer (2015). Xcodeghost used unprecedented infection strategy against apple. http://www.computerworld.com/article/2986768/application-development/xcodeghost-used-unprecedented-infection-strategy-against-apple.html. Online; accessed October 13, 2015.

[Hsu et al., 2006] Hsu, F., Chen, H., Ristenpart, T., Li, J., and Su, Z. (2006). Back to the future: A Framework for Automatic Malware Removal and System Repair. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 257–268. IEEE.

[Hudson, 2015] Hudson, T. (2015). Thunderstrike 2 - trammell hudson's projects. https://trmm.net/Thunderstrike_2. Online; accessed November 13, 2015.

[Jain and Sekar, 2000] Jain, K. and Sekar, R. (2000). User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement. In *NDSS*.

[Jana et al., 2011] Jana, S., Porter, D. E., and Shmatikov, V. (2011). TxBox: Building Secure, Efficient Sandboxes with System Transactions. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 329–344. IEEE.

[jduck, 2014] jduck (2014). CVE-2010-3338 Windows Escalate Task Scheduler XML Privilege Escalation — Rapid7. http://www.rapid7.com/db/modules/exploit/windows/local/ms10_092_schelevator. Online; accessed September 18, 2015.

[Kaspersky Lab] Kaspersky Lab. What is Flame Malware? http://www.kaspersky.com/flame. Online; accessed March 4, 2016.

[Katcher, 1997] Katcher, J. (1997). Postmark: A new file system benchmark. Technical Report TR3022, Network Appliance.

[Kim, 2011] Kim, A. (2011). Security Researcher Reveals iOS Security Flaw, Gets Developer License Revoked. http://www.macrumors.com/2011/11/08/

security-researcher-reveals-ios-security-flaw-gets-developer-license-revoked/. Online; accessed November 12, 2015.

[Kornblum, 2006] Kornblum, J. (2006). Identifying Almost Identical Files Using Context Triggered Piecewise Hashing. *Digital investigation*, 3:91–97.

[Krohn et al., 2007] Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M. F., Kohler, E., and Morris, R. (2007). Information Flow Control for Standard OS Abstractions. In *SOSP*.

[Leonard et al., 2015] Leonard, T. et al. (2015). 0install: Overview. http://0install.net/. Online; accessed September 18, 2015.

[letiemble, 2011] letiemble (2011). Mac OS X Application SandBoxing: what about Apple ? http://blog.laurent.etiemble.com/index.php?post/2011/11/06/About-Mac-OS-X-Application-SandBoxing. Online; accessed November 12, 2015.

[Li, 2015] Li, H. (2015). CVE-2015-0016: Escaping the Internet Explorer Sandbox. http://blog.trendmicro.com/trendlabs-security-intelligence/cve-2015-0016-escaping-the-internet-explorer-sandbox. Online; accessed September 18, 2015.

[Li et al., 2007] Li, N., Mao, Z., and Chen, H. (2007). Usable Mandatory Integrity Protection for Operating Systems . In *S&P*.

[Liang et al., 2009] Liang, Z., Sun, W., Venkatakrishnan, V. N., and Sekar, R. (2009). Alcatraz: An Isolated Environment for Experimenting with Untrusted Software. In *TISSEC*.

[Liang et al., 2003] Liang, Z., Venkatakrishnan, V., and Sekar, R. (2003). Isolated program execution: An application transparent approach for executing untrusted programs. In *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, pages 182–191. IEEE.

[Ligatti et al., 2005] Ligatti, J., Bauer, L., and Walker, D. (2005). Edit Automata: Enforcement Mechanisms for Run-Time Security Policies. *International Journal of Information Security*, 4(1-2):2–16.

[Lin, 2013] Lin, L. (2013). Gatekeeper on Mac OS X 10.9 Mavericks. https://blog.trendmicro.com/trendlabs-security-intelligence/gatekeeper-on-mac-os-x-10-9-mavericks/. Online; accessed November 12, 2015.

[Linux Kernel Organization, 2015] Linux Kernel Organization, I. (2015). SECure COMPuting with filters. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt. Online; accessed September 18, 2015.

[Loscocco and Smalley, 2001a] Loscocco, P. and Smalley, S. (2001a). Integrating Flexible Support for Security Policies into the Linux Operating System. In *USENIX ATC*.

[Loscocco and Smalley, 2001b] Loscocco, P. and Smalley, S. (2001b). Meeting Critical Security Objectives with Security-Enhanced Linux. In *Ottawa Linux Symposium*.

[Lu et al., 2010] Lu, L., Yegneswaran, V., Porras, P., and Lee, W. (2010). Blade: an attack-agnostic approach for preventing drive-by malware infections. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 440–450. ACM.

[Mao et al., 2011] Mao, Z., Li, N., Chen, H., and Jiang, X. (2011). Combining Discretionary Policy with Mandatory Information Flow in Operating Systems. In *TISSEC*.

[McAfee Labs, 2015] McAfee Labs (2015). McAfee Labs Threats Report August 2015. http://www.mcafee.com/us/resources/reports/rp-quarterly-threats-aug-2015.pdf. Online; accessed September 18, 2015.

[Microsoft, 2015a] Microsoft (2015a). URL Security Zones (Windows) - MSDN - Microsoft. https://msdn.microsoft.com/en-us/library/ie/ms537021%28v=vs.85%29.aspx. Online; accessed September 18, 2015.

[Microsoft, 2015b] Microsoft (2015b). What is Protected View? - Office Support. https://support.office.com/en-au/article/What-is-Protected-View-d6f09ac7-e6b9-4495-8e43-2bbcdbcb6653. Online; accessed September 18, 2015.

[Microsoft, 2015c] Microsoft (2015c). What is the Windows Integrity Mechanism? https://msdn.microsoft.com/en-us/library/bb625957.aspx. Online; accessed September 18, 2015.

[Microsoft, 2015d] Microsoft (2015d). Working with the AppInit_DLLs registry value. http://support.microsoft.com/kb/197571. Online; accessed September 18, 2015.

[Microsoft Research, 2015] Microsoft Research (2015). Detours. http://research.microsoft.com/en-us/projects/detours/. Online, accessed September 18, 2015.

[Mital, 2010] Mital, B. (2010). *A Framework for Enforcing Information Flow Policies*. PhD thesis, The Graduate School, Stony Brook University: Stony Brook, NY.

[Mozai, 2013] Mozai (2013). [ubuntu] AppArmor preventing me from using video chat. http://ubuntuforums.org/showthread.php?t=2100980. Online; accessed November 12, 2015.

[Mozilla, 2015] Mozilla (2015). Buildbot/Talos/Tests. https://wiki.mozilla.org/Buildbot/Talos/Tests. Online; accessed September 18, 2015.

[Mozilla Developer Network, 2015] Mozilla Developer Network (2015). Window.postmessage(). https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage. Online; accessed April 7, 2016.

[Myers and Liskov, 1997] Myers, A. C. and Liskov, B. (1997). *A Decentralized Model for Information Flow Control*, volume 31. ACM.

[Myers et al., 2001] Myers, A. C., Zheng, L., Zdancewic, S., Chong, S., and Nystrom, N. (2001). Jif: Java Information Flow. *Software release. Located at http://www. cs. cornell. edu/jif*, 2005.

[NTT DATA Corporation, 2010] NTT DATA Corporation (2010). Akari. http://akari.sourceforge.jp/. Online; accessed February 20, 2014.

[Offensive Security, 2014] Offensive Security (2014). Exploits Database. http://www.exploit-db.com/. Online; accessed November 11, 2014.

[Packet Storm, 2015] Packet Storm (2015). Packet Storm. http://packetstormsecurity.com. Online; accessed September 18, 2015.

[Padala, 2002] Padala, P. (2002). Playing with ptrace, Part I. www.linuxjournal.com/article/6100. Online; accessed September 18, 2015.

[Parampalli et al., 2008] Parampalli, C., Sekar, R., and Johnson, R. (2008). A Practical Mimicry Attack Against Powerful System-Call Monitors. In *ASIACCS*.

[Porter et al., 2014] Porter, D. E., Bond, M. D., Roy, I., Mckinley, K. S., and Witchel, E. (2014). Practical Fine-Grained Information Flow Control Using Laminar. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(1):4.

[Porter et al., 2011] Porter, D. E., Boyd-Wickizer, S., Howell, J., Olinsky, R., and Hunt, G. C. (2011). Rethinking the Library OS from the Top Down. *SIGARCH Comput. Archit. News*, 39(1):291–304. Available from: http://doi.acm.org/10.1145/1961295.1950399.

[Porter et al., 2009] Porter, D. E., Hofmann, O. S., Rossbach, C. J., Benn, A., and Witchel, E. (2009). Operating System Transactions. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 161–176, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/1629575.1629591.

[Potter and Nieh, 2010] Potter, S. and Nieh, J. (2010). Apiary: Easy-to-Use Desktop Application Fault Containment on Commodity Operating Systems. In *USENIX conference on USENIX annual technical conference*.

[Provos, 2003] Provos, N. (2003). Improving Host Security with System Call Policies. In *USENIX Security*.

[Provos et al., 2003] Provos, N., Markus, F., and Peter, H. (2003). Preventing Privilege Escalation. In *USENIX Security*.

[Rahul Kashyap, 2013] Rahul Kashyap, R. W. (2013). Application Sandboxes: A Pen-Tester's Perspective. http://labs.bromium.com/2013/07/23/application-sandboxes-a-pen-testers-perspective/. Online; accessed September 23, 2015.

[Reddit Discussion, 2014] Reddit Discussion (2014). ELI5: Why don't people use the Mac App Store? https://www.reddit.com/r/apple/comments/2npwfp/eli5_why_dont_people_use_the_mac_app_store/. Online; accessed November 12, 2015.

[Reis and Gribble, 2009] Reis, C. and Gribble, S. D. (2009). Isolating Web Programs in Modern Browser Architectures. In *EuroSys*.

[Roesner et al., 2012] Roesner, F., Kohno, T., Moshchuk, A., Parno, B., Wang, H. J., and Cowan, C. (2012). User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 224–238. IEEE.

[Sandboxie Holdings, LLC., 2015] Sandboxie Holdings, LLC. (2015). Sandboxie. http://www.sandboxie.com/. Online; accessed September 18, 2015.

[Schneider, 2000] Schneider, F. B. (2000). Enforceable Security Policies. In *TISSEC*.

[Seaborn, 2015] Seaborn, M. (2015). Plash. http://plash.beasts.org/contents.html. Online; accessed October 14, 2015. Available from: http://plash.beasts.org.

[Sekar et al., 2003] Sekar, R., Venkatakrishnan, V., Basu, S., Bhatkar, S., and DuVarney, D. C. (2003). Model-Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications. In *SOSP*.

[Sun et al., 2005] Sun, W., Liang, Z., Venkatakrishnan, V. N., and Sekar, R. (2005). One-Way Isolation: An Effective Approach for Realizing Safe Execution Environments. In *NDSS*.

[Sun et al., 2008a] Sun, W., Sekar, R., Liang, Z., and Venkatakrishnan, V. N. (2008a). Expanding Malware Defense by Securing Software Installations. In *DIMVA*.

[Sun et al., 2008b] Sun, W., Sekar, R., Poothia, G., and Karandikar, T. (2008b). Practical Proactive Integrity Preservation: A Basis for Malware Defense. In *S&P*.

[Sze et al., 2014] Sze, W. K., Mital, B., and Sekar, R. (2014). Towards More Usable Information Flow Policies for Contemporary Operating Systems. In *SACMAT*.

[Sze and Sekar, 2013] Sze, W. K. and Sekar, R. (2013). A Portable User-Level Approach for System-wide Integrity Protection. In *ACSAC*.

[Tanase, 2015] Tanase, S. (2015). Satellite Turla: APT Command and Control in the Sky. https://securelist.com/blog/research/72081/satellite-turla-apt-command-and-control-in-the-sky/. Online; accessed March 4, 2016.

[The Volatility Foundation, 2015] The Volatility Foundation (2015). Volatility foundation. http://www.volatilityfoundation.org/. Online; accessed October 13, 2015.

[Tiwari et al., 2012] Tiwari, M., Mohan, P., Osheroff, A., Alkaff, H., Shi, E., Love, E., Song, D., and Asanović, K. (2012). Context-centric security. In *Proceedings of the 7th USENIX conference on Hot Topics in Security*, pages 9–9. USENIX Association.

[Tsai et al., 2014] Tsai, C.-C., Arora, K. S., Bandi, N., Jain, B., Jannen, W., John, J., Kalodner, H. A., Kulkarni, V., Oliveira, D., and Porter, D. E. (2014). Cooperation and security isolation of library oses for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 9:1–9:14, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/2592798.2592812.

[Ubuntu, 2015] Ubuntu (2015). AppArmor. https://wiki.ubuntu.com/AppArmor/. Online; accessed September 23, 2015.

[Wahbe et al., 1993] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. (1993). Efficient Software-based Fault Isolation. *SIGOPS Oper. Syst. Rev.*, 27(5):203–216. Available from: http://doi.acm.org/10.1145/173668.168635.

[Wang et al., 2007] Wang, H. J., Fan, X., Howell, J., and Jackson, C. (2007). Protection and communication abstractions for web browsers in mashupos. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 1–16. ACM.

[Wang et al., 2013] Wang, T., Lu, K., Lu, L., Chung, S., and Lee, W. (2013). Jekyll on iOS: When Benign Apps Become Evil. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 559–572, Berkeley, CA, USA. USENIX Association. Available from: http://dl.acm.org/citation.cfm?id=2534766.2534814.

[Wang Jiaye, 2011] Wang Jiaye, H. C. (2011). Improve cross-domain communication with client-side solutions. http://www.ibm.com/developerworks/library/wa-crossdomaincomm/. Online; accessed March 25, 2016.

[Ward, 2014] Ward, S. (2014). iSIGHT discovers zero-day vulnerability CVE-2014-4114 used in Russian cyber-espionage campaign. http://www.isightpartners.com/2014/10/cve-2014-4114/. Online; accessed September 18, 2015.

[Watson et al., 2003] Watson, R., Morrison, W., Vance, C., and Feldman, B. (2003). The TrustedBSD MAC Framework: Extensible Kernel Access Control for FreeBSD 5.0. In *USENIX Annual Technical Conference, FREENIX Track*, pages 285–296.

[Wei et al., 2012] Wei, X., Gomez, L., Neamtiu, I., and Faloutsos, M. (2012). Permission Evolution in the Android Ecosystem. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 31–40, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/2420950.2420956.

[Wright et al., 2002] Wright, C., Cowan, C., Smalley, S., Morris, J., and Kroah-Hartman, G. (2002). Linux Security Modules: General Security Support for the Linux Kernel. In *USENIX Security*.

[Xing et al., 2015] Xing, L., Bai, X., Li, T., Wang, X., Chen, K., Liao, X., Hu, S.-M., and Han, X. (2015). Cracking App Isolation on Apple: Unauthorized Cross-App Resource Access on MAC OS X and iOS. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 31–43, New York, NY, USA. ACM. Available from: http://doi.acm.org/10.1145/2810103.2813609.

[XSB Research Group, 2012] XSB Research Group (2012). XSB. http://xsb.sourceforge.net/. Online; accessed April 4, 2016.

[Yee et al., 2009] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Orm, T., Okasaka, S., Narula, N., Fullagar, N., and Inc, G. (2009). Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *S&P*.

[Zeldovich et al., 2006] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazières, D. (2006). Making Information Flow Explicit in HiStar. In *OSDI*.