# Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable

by

Richard Ta-Min

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

Copyright © 2006 by Richard Ta-Min

# Abstract

Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable

Richard Ta-Min

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2006

In current commodity systems, applications have no way of limiting their trust in the underlying operating system (OS), leaving them at the complete mercy of an attacker who gains control over the OS. In this work, we describe the design and implementation of Proxos, a system that allows applications to configure their trust in the OS by partitioning the system call interface into trusted and untrusted components. System call routing rules that indicate which system calls are to be handled by the untrusted commodity OS, and which are to be handled by a trusted *private OS*, are specified by the application developer. We find that rather than defining a new system call interface, routing system calls of an existing interface allows applications currently targeted towards commodity operating systems to isolate their most sensitive components from the commodity OS with only minor source code modifications.

We have built a prototype of our system on top of the Xen Virtual Machine Monitor with Linux as the commodity OS. In practice, we find that the system call routing rules are short and simple – on the order of 10's of lines of code. In addition, applications in Proxos incur only modest performance overhead, with most of the cost resulting from inter-VM context switches.

# Acknowledgements

First and foremost, I would like to thank Professor David Lie for his patient supervision, his financial support and the large amount of time we spent discussing this research.

I am thankful to Lionel Litty, with whom I worked together on many aspects of this project. I would also like to thank Tom Hart, Ian Sin and Jesse Pool for their feedback during our numerous meeting groups.

I am also grateful to my family for their moral support.

Finally, I would like to thank the Edward S. Rogers Sr. Ontario Graduate Scholarship fund as well as the department of Electrical and Computer Engineering for their financial support.

# Contents


1 Introduction 1

2 Background 5

2.1 Definition . . . . . . . . . . 5

2.2 Hardware virtualization . . . . . . . . . . 6

2.2.1 Virtualizing memory . . . . . . . . . . 6

2.2.2 Virtualizing devices . . . . . . . . . . 7

2.2.3 Virtualizing the CPU, interrupts and exceptions . . . . . . . . . . 7

2.3 Security properties of VMMs . . . . . . . . . . 8

2.3.1 Strong isolation property . . . . . . . . . . 8

2.3.2 High security assurance . . . . . . . . . . 8

2.4 VMMs in security architectures . . . . . . . . . . 9

2.5 VMMs in non-security architectures . . . . . . . . . . 10

2.6 VMMs and microkernels . . . . . . . . . . 10

3 Overview 12

3.1 System Architecture . . . . . . . . . . 12

3.2 Security Guarantees . . . . . . . . . . 15

3.3 The Proxos Routing Language . . . . . . . . . . 16

4 Implementation 19

4.1 Modifications to the VMM and the Commodity OS . . . . . . . . . . . . 20
4.2 The Proxos Prototype . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.3 Private OS Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
4.4 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

**5 Applications** **29**
5.1 Secure Web Browser . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.2 SSH Authentication Server . . . . . . . . . . . . . . . . . . . . . . . . 30
5.3 SSL Certificate Service and Apache . . . . . . . . . . . . . . . . . . . 33
5.4 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

**6 Evaluation** **35**
6.1 Microbenchmarks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
6.2 Application Benchmarks . . . . . . . . . . . . . . . . . . . . . . . . . . 37

**7 Related work** **40**
7.1 Systems with flexible architectures . . . . . . . . . . . . . . . . . . . . 40
7.2 Operating system security . . . . . . . . . . . . . . . . . . . . . . . . . 41
7.3 Systems that provide isolation . . . . . . . . . . . . . . . . . . . . . . . 43

**8 Conclusion** **46**

**Bibliography** **48**

# List of Tables

4.1 Number of lines of code in each component in our Proxos prototype . . . 28

5.1 Size of routing rules and number of LOC modified for each application . 34

6.1 Forwarded system call latencies on LMbench microbenchmarks . . . . . . 37

# List of Figures

3.1 The Proxos system architecture . . . . . . . . . . . . . . . . . . . . . . . . 13

3.2 Routing example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

4.1 Private application start-up sequence . . . . . . . . . . . . . . . . . . . . 20

5.1 Comparison of the original SSH server and our private SSH server . . . . 32

6.1 Breakdown of costs incurred in a forwarded system call . . . . . . . . . . 36

6.2 SSH benchmark . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

# List of Acronyms

| Acronym | Definition |
|---|---|
| IDS | Intrusion Detection System |
| IPC | Inter-Process Communication |
| NIC | Network Interface Card |
| OS | Operating System |
| RPC | Remote Procedure Call |
| SSH | Secure Shell |
| SSL | Secure Socket Layer |
| TCB | Trusted Computing Base |
| TLB | Translation Lookaside Buffer |
| VM | Virtual Machine |
| VMM | Virtual Machine Monitor |

# Chapter 1

# Introduction

While significant effort has been invested into making our computing infrastructure more secure, the number of security incidents continues to increase at an alarming pace. The CERT Coordination Center reports that the number of security incidents increased approximately six-fold in the three years between 2000 and 2003, after which they indicate that incidents had become so commonplace that they were not even worth reporting [5]. Despite these statistics, businesses and individuals continue to put increasing trust in computers to store and secure sensitive information, such as financial data, health records, and recently, votes for government elections [22].

Though a great deal of work goes into making operating system kernels more secure, in the vast majority of cases the vulnerabilities being exploited are not in the kernel, but in privileged applications running as user processes. The problem lies not in the reliability of kernel code, but in the overly permissive interface that commodity operating systems (OSs) export, which a privileged application can abuse to make the operating system kernel read or modify the state of any other application. On the other hand, many applications require such privileges to run on a commodity operating system, providing the attacker with many opportunities to take control of the operating system interface. As a result, it seems appropriate that applications that perform security-sensitive operations

should have little or no trust in the kernel that lies on the other side of a commodity OS interface.

There have been several attempts to address this situation. One solution is to use a microkernel [1], which minimizes the amount of code running in supervisor mode. However, changing the underlying architecture of the OS kernel without changing the interface that applications use will not give applications any more protection than they currently have. On the other hand, narrowing the application-OS interface requires a large amount of effort to port or rewrite applications currently targeted towards a broad commodity OS interface [35]. There have also been attempts to restrict the interface in existing commodity OSs such as Linux with fine-grained access controls [25]. While effective in principle, the ability to have such controls means that the policy description must be equally fine-grained, making it very complex and time consuming to configure such systems correctly [19]. A third solution is to run the security-sensitive application in its own *private OS* on a virtual machine (VM) executing on top of a virtual machine monitor (VMM), and thus completely remove all other applications from the trusted computing base (TCB) of the system [13]. This private OS would only support the one application and be specially tailored to its needs. The problem that arises is that applications typically share data and interact with other applications through operating system facilities such as files and pipes. Therefore, short of changing the way applications communicate, we are forced to move the other applications into the private OS as well. As a result, the security-sensitive application is made to tolerate other applications in its TCB that it needs to interact with, but does not necessarily trust.

In this work, we attempt to address these issues by building a system that allows an application developer to choose what operating system facilities should be provided by an untrusted commodity OS, and what facilities need to be provided by a trusted private OS. In this way, applications may continue to use functionality in the commodity OS to communicate with other programs, and avoid having to duplicate functionality in the

private OS that does not have to be trusted. This ability is provided by running both commodity and private OSs on a VMM, and using a thin operating system proxy, called *Proxos*, which we have designed. Proxos is a small library that mimics an operating system by handling system calls made by the application.

Proxos takes a novel approach to allowing applications to specify their trust in an operating system. Rather than requiring that the application developer partition the application code into components based on whether they trust the commodity OS or not [35], Proxos only requires the developer to partition the system call interface into system calls that must be trusted and those that need not. Using high-level system call routing rules specified by the application developer, Proxos transparently routes each system call made by the application to the commodity OS if the request does not need to be trusted, or to the private OS if it does. Specifying trust by partitioning the system call interface has the benefit that applications currently implemented for commodity OSs can be easily ported to Proxos with very little effort (typically by only modifying on the order of several hundred lines of code). Consequently, the application developer is able to remove the entire commodity OS from the TCB of their application while maintaining reasonable performance.

In this research, we make three main contributions. First, we have designed a language that allows developers to configure trust relationships using short and simple system call routing rules. In practice, we find that routing rules can usually be specified in 50 lines or less. Second, we have designed and implemented a prototype of Proxos on top of the Xen VMM [3], with Linux as the commodity OS. We describe the modifications we made to Xen and Linux and evaluate the amount of code that these modifications add to each component. Finally, we demonstrate the utility of our system by porting three existing applications: a web browser that protects user privacy, an SSH authentication server, and an SSL certificate service used by the Apache web server. We describe the security of the new applications, the issues we encountered in porting them to Proxos,

as well as the performance impact of moving to Proxos. This project [36] also appears in the Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, November 2006 (OSDI 2006). The author contributed to the novelty of the system and was also responsible for most of the implementation.

The next chapter will give some background on VMMs and how they are used in security architectures. Next, we will give a high-level description of the system architecture needed to run Proxos applications, as well as a description of the Proxos routing language in Chapter 3. Chapter 4 follows with an explanation of our prototype, and gives details on modifications we made to Xen and Linux, details on our Proxos implementation, and details on some example private OS functions we have written. To show what applications one might run on Proxos, we describe three representative applications we have ported to Proxos in Chapter 5, and evaluate the performance impact of Proxos against a vanilla Xen/Linux system in Chapter 6. Finally, we finish with related work in Chapter 7, and give our conclusions in Chapter 8.

# Chapter 2

# Background

This chapter gives an overview of virtual machine monitors (VMM). It first gives a definition of a VMM followed by an overview of how virtualization is implemented. Next, the security characteristics of a VMM are explored and the application of VMMs in some recent projects is described. Finally, we conclude this chapter with a brief comparison between VMMs and microkernels.

## 2.1 Definition

A virtual machine monitor (VMM) is a thin software layer that sits between an operating system (OS) and the hardware. The VMM virtualizes the physical hardware into virtual machine (VM) compartments. An operating system that executes inside a virtual machine has the illusion that it is running directly on the bare hardware. VMMs typically virtualize a physical host into multiple VM compartments each running a distinct OS instance that we call a *guest OS*. Guest OSs are oblivious of other guest OSs running on the same physical host. They have the illusion that they are in complete control of the underlying hardware, they run independently and are isolated from one another. VMMs such as Xen [3] can concurrently execute multiple guest OSs such as Linux and Windows XP.

## 2.2 Hardware virtualization

It is the VMM's responsibility to virtualize the underlying hardware and to multiplex the resources across the guest OSs. For instance, the physical hard drive could be divided into smaller virtual subdisks and each guest OS is assigned to one subdisk. In another example, the network interface card (NIC) is virtualized into several virtual NICs which are then assigned to each guest OS. In addition, the physical memory of the system has to be subdivided into distinct memory regions and the VMM must constrain each guest OS to its own physical memory region.

In order to virtualize the hardware and confine a guest OS within its VM compartment, the VMM executes at the processor's highest privilege level. The VMM validates and interposes on all privileged instructions that the guest OS executes to change the state of the underlying hardware. However, virtualization is expensive without proper hardware support. Some architectures, such as the Intel x86 architecture, are known to for their lack of virtulization support [30]. For instance, the x86 instruction set contains a subset of privilege instructions that fail silently instead of raising an exception when executed from a low privilege level. To improve the performance of VMMs, modern VMMs take a number of different approaches. VMWare ESX [38] performs dynamic binary rewriting of the OS binary image to convert non-virtualizable instructions into virtualizable ones. Xen [3] uses an invasive approach called paravirtualization which involves modifying the guest OS. The operating system source code is rewritten and optimized to run within the virtual machine environment.

The following sections describe how the various system resources are virtualized.

### 2.2.1 Virtualizing memory

Guest OSs must be constrained to use their allocated memory and the VMM must prevent one guest OS from tampering with the memory allocated to other VMs. To

virtualize memory, the VMM must validate and interpose on all privileged instructions that manipulate the page tables and translation lookaside buffers (TLB) of the processor's memory management unit (MMU). *Shadow* page tables [7] is a commonly employed technique whereby the guest OS is allowed to manipulate a set of fake page tables that are made invisible to the processor's MMU. The VMM is then responsible for validating the entries of the fake page tables and to propagate the values to the MMU-visible *shadow* page tables. Other VMMs such as Xen [3] rely on paravirtualization and employ a different approach. The source code of the guest OS is modified so that all page table modifications are made by first trapping into the VMM. The VMM then validates the entries before applying the changes.

### 2.2.2 Virtualizing devices

To multiplex I/O devices among multiple guest OSs, VMMs typically split device drivers into a frontend interface and a backend driver. The backend driver has direct access to the physical device and multiplexes the I/O requests from the guest OSs. The backend driver typically executes inside the VMM or in a special VM compartment reserved for hosting device drivers [23, 12]. The frontend interface could be either a special driver that runs inside the guest OS or a virtualized interface that emulates the physical device. The main function of the frontend interface is to forward I/O requests made by the guest OS to the backend driver.

### 2.2.3 Virtualizing the CPU, interrupts and exceptions

VMMs share the CPU among guest OSs in a way similar to the way conventional OSs share the CPU among several user-space processes. The VMM keeps track of the execution state of each guest OS and schedules the CPU among the VMs.

Virtualizing interrupts and exceptions is also done in a straightforward manner. Since the VMM executes at the highest privilege level of the processor, it receives all interrupts

and exceptions that occur on the system. When an interrupt is raised, the VMM records the interrupt and if necessary, it can jump to the interrupt handler routine inside the appropriate guest OS.

## 2.3 Security properties of VMMs

This section describes the properties of VMMs that make them appealing to the implementation of security architectures.

### 2.3.1 Strong isolation property

As seen from the previous section, a VMM maintains a strong isolation across the VM compartments. Each guest OS is assigned its own private set of virtual resources: virtual devices, virtual CPU and physical memory. In addition, by validating and interposing on resource access, the VMM prevents one guest OS from tampering with the resources of another VM. This strong isolation property is an important aspect in the implementation of secure systems as the effect of a compromise of an application is contained within the confines of a VM compartment and does not affect the whole system.

VMMs pave the way for the design of more robust security mechanisms. Traditional security mechanisms that are implemented on top of conventional operating systems typically rely on the integrity of the kernel and are rendered useless if the OS kernel is compromised. VMMs allow security mechanisms to be implemented outside of a guest OS and to be resistant to a full compromise of the guest OS.

### 2.3.2 High security assurance

The main function of the VMM is to isolate and multiplex the hardware resources among the VM compartments. As a result, the code size of a VMM is typically simpler and smaller than conventional monolithic operating systems. The inclusion of device drivers

inside the VMM could potentially increase the complexity and size of the VMM. However, Fraser et al. [12] and LeVasseur et al. [23] demonstrate that untrusted device drivers could be safely implemented outside of the VMM code base.

In addition to its small code size, a VMM only exports a very narrow interface to the guest OSs. This interface is generally restricted to the isolation and allocation of resources. The small code size reduces the risk of the VMM containing an exploitable bug and the narrow interface limits number of ways a malicious guest OS could exploit a bug inside the VMM. Those two factors combined allow a VMM to provide for a high level of security assurance.

## 2.4 VMMs in security architectures

Since the VMM is unlikely to be subverted, one can trust the VMM to always enforce the strong isolation across the VMs running on the same physical host. This section reviews some recent security-related projects that leverage the isolation and high assurance of a VMM. Terra [13] uses a VMM to execute multiple applications with diverging security requirements on the same physical machine by isolating each security-sensitive application in their separate VM. Collapsar [20] and Potemkin [37] use VMMs to host high-interaction honeypot farms to study malicious attackers. VMMs allow them to host multiple honeypots on a few physical hosts, which facilitates deployment and administration, while maintaining strong isolation between the honeypots. Traditional intrusion detection systems (IDS) that are implemented on a conventional operating system are vulnerable if the underlying OS kernel is compromised. Asrigo et al. [2] use a VMM to host the IDS outside of the guest OS making the system resistant against a full operating system compromise.

## 2.5 VMMs in non-security architectures

Although VMMs are very appealing for implementing secure systems, they are also used in other non-security-related systems. A VMM provides a clean separation between hardware and software by encapsulating all the software state of an OS and its applications within a VM. This allows for the efficient live migration [6] of an entire OS between physical hosts with minimal downtime. A VMM is logically located below the OS. Hence, it allows for the complete visibility of what is going on inside the OS. TTVM [21] uses a VMM to help developers efficiently debug operating systems. It allows for the deterministic replay of OS execution and the insertion of breakpoints to pause the execution of the OS.

## 2.6 VMMs and microkernels

In this section, we give an overview of microkernels and provide a brief comparison between VMM and microkernel architectures. Traditional OS kernels are typically large and include many services. Such OS kernels, known as monolithic kernels, are difficult to maintain and extend. Microkernels [24] were developed as a response to improve the design of monolithic kernels. The microkernel approach tries to keep the kernel as small as possible by implementing many of the services commonly provided by a monolithic OS outside of the kernel. Services such as the file system, networking, device drivers and even memory paging are all implemented outside of the kernel in separate protection domains. The microkernel itself is restricted to provide one primitive mechanism: inter-process communication (IPC) among the protection domains.

VMMs and microkernels share similar goals [14] as they both try to refactor systems into separate components. However, the two architectures differ in a number of ways. First, while IPC is a fundamental mechanism for microkernels to enable controlled communication between protection domains, VMMs instead focus on maintaining isolation

between VMs. Hence, IPC between VMs is not part of the design of pure VMM architectures. A more important difference between VMMs and microkernels is the granularity at which the two architectures try to divide the system into separate compartments. Microkernels focus on dividing the functional units of a monolithic operating system into separate protection domains and export to each domain the narrow IPC interface of the microkernel. In contrast, VMMs execute entire guest OSs in separate VMs and export to each guest OS a virtualized interface of the underlying hardware.

Despite the differences, the distinction between VMMs and microkernels has often been blurred as numerous projects deviate from pure-VMM or pure-microkernel designs to offer hybrid solutions. For instance, the L4 microkernel has been adapted to execute an entire Linux OS within one protection domain [15]. Another example, Fraser et al. [12] isolate device drivers in separate VMs and extend the Xen VMM with IPC facilities to give guest OSs access to the device drivers. Finally, in this research project we used a VMM to implement our prototype and we extended the VMM with IPC facilities to allow guest OSs to make remote procedure calls (RPC) to other guest OSs. However, we want to point out that nothing in this work is VMM-specific. We believe that a modified microkernel could also have been used as our experimental substrate.

# Chapter 3

# Overview

In this chapter, we describe the overall architecture of the system, as well as a description of the security guarantees our system provides. Then, we give a description of the Proxos system call routing language.

## 3.1 System Architecture

The architecture of our system is illustrated in Figure 3.1. The system consists of several VMs running on top of a VMM that enforces memory isolation between the VMs and allocates CPU execution time to the VMs. VMs can make *hypercalls* to the underlying VMM to access resources such as disks and other devices, or to signal or create other VMs. A *commodity OS VM* runs a commodity OS that provides the facilities usually found in a standard operating system, such as file system implementations, a network stack and a user interface. An *administrative VM* (not shown in the diagram) contains management tools used to create and manage other VMs. Applications that want to be isolated from the commodity OS are run inside their own *private VM* along with a Proxos instance. We call such applications *private applications*. A set of methods inside the private VM implement a *private OS*, whose purpose is to handle system calls the private application does not trust the commodity OS with.

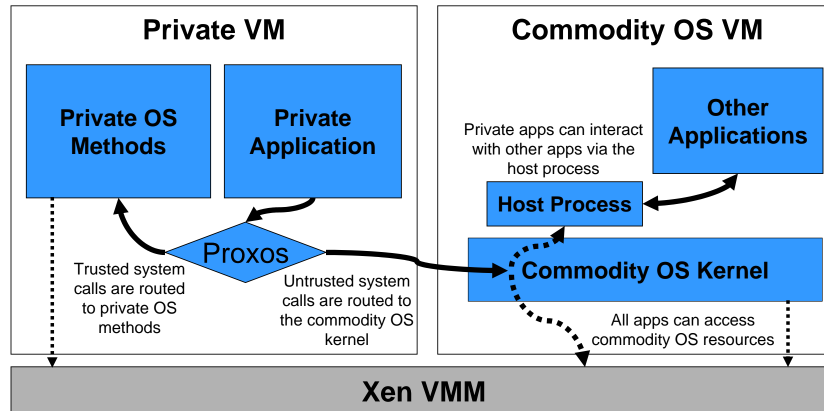


Figure 3.1: The Proxos System Architecture. Proxos handles all system calls the private application makes by routing them to either the commodity OS or the private OS.

Proxos handles all system calls made by the application. Depending on the routing rules configured by the application developer, Proxos will route non-security-sensitive system calls to the commodity OS via inter-VM remote procedure calls (RPCs), and security-sensitive system calls to methods in the private OS. Both Proxos and the private OS are implemented as libraries that are statically linked with the application. As a result, all system calls are converted into subroutine calls to Proxos. The application, along with Proxos and the private OS run on the bare VMM. Since only one application runs in each private VM, all code in a private VM runs in the same protection domain.

To run a commodity application as a private application, the developer first identifies which operating system objects the application uses and that need to be protected from a compromised commodity OS. With this knowledge, the developer identifies the system calls that access these objects and specifies that they are to be forwarded to the private OS using the routing language described in Section 3.3. The private OS methods can be implemented especially for the application by the developer, or even obtained from a library of generic private OS methods provided by a third-party. Section 4.3 describes some private OS methods that we have implemented.

The developer may then have to perform application source code modifications in

order to compile it statically, and have it use the facilities that Proxos provides. However, since Proxos exports the same system call interface as the commodity OS, these changes are generally minor. For instance, we were able to port the Glibc library (version 2.3.3) to Proxos with only 218 lines of source code modifications. Next, the private application, the private OS methods, the routing rules and Proxos are all compiled into a single binary, which can be loaded into an empty VM. The developer gives this binary image to the VMM administrator, who registers the new private application with the VMM using the administrative VM. Because the private application binaries are stored directly on the VMM, they are safe from tampering by an adversary who has subverted the commodity OS.

To run a private application, a user on the commodity OS invokes a *host process*, which requests the VMM to instantiate a new VM containing the private application. From this point on, the host process becomes the embodiment of the private application on the commodity OS. The commodity OS attributes any forwarded system call it receives from the private application to the host process that instantiated it. The commodity OS uses the user ID of this host process to make decisions about what operating system objects (such as files or sockets) the application is allowed to access, and also attributes resources used by the forwarded system calls to the host process. In this way, the commodity OS ensures fairness and security between requests made by private applications and requests made by applications running natively on the commodity OS.

Through its host process, a private application can interact with other applications running in the commodity OS by using facilities provided by the commodity OS. For example, by configuring Proxos to forward `mknod` and `open` system calls to the commodity OS, a private application can create a named pipe between it and a commodity OS application. Then, by routing `read` and `write` system calls to the commodity OS, it can communicate with the commodity OS application by making those system calls on the named pipe. For a communication channel to be created, cooperation is required from

both applications, who must agree to communicate, and from the commodity OS, who must agree to fulfill the system call requests made by both applications.

## 3.2 Security Guarantees

While the commodity OS may at some point become under the complete control of an attacker, we assume that the underlying VMM cannot be subverted and that it continues to enforce isolation between VMs. We also rely on the application developer to properly specify what sensitive components of the interface between the application and the operating system must be protected from the commodity OS. Based on these assumptions, our system maintains the confidentiality and integrity of sensitive private application data even in the face of a compromised commodity OS. The isolation property of the VMM prevents the compromised OS from directly interfering with the private application. The compromised commodity OS can only tamper with system calls that are routed to it by Proxos. However, since these system calls were identified as non-security-critical by the developer, the compromised OS should not be able to affect the private application in any security-critical way. We point out that if the routing rules are specified incorrectly, or if a bug in the application causes it to send sensitive data to an interface that the developer believes should have only held non-sensitive data, then sensitive data could be leaked to the commodity OS. In addition, while the confidentiality and integrity of sensitive private application data are maintained, a compromised OS can impact the availability of a private application by not performing the system calls that are forwarded to it.

So far, we have considered protecting the private application from a potentially malicious OS. However, one could envision the case of a buggy private application that could negatively affect the commodity OS through the system calls it forwards to the OS. However, our design restricts the capabilities of the private application within the commodity OS to that of its host process. Since the private application only has the

rights of the user who invoked it, our system does not weaken any existing mechanism that guarantees fairness among users and processes running on the commodity OS.

## 3.3 The Proxos Routing Language

Proxos may route each invocation of a particular system call differently depending on rules specified by the application developer. For example, Proxos may route `read` system calls differently depending on what file is being read. We wish to provide a simple and intuitive way for an application developer to partition the system call interface. In principle, one could specify a routing rule for each of the over 200 system calls that a commodity OS like Linux provides, but this would be complex and time consuming. Further, we do not believe it necessary in most cases to have such fine-grained control over system call routing. We organize system calls by the resources they access and create a Proxos routing language with which the developer can specify routes for those resources. In this language, the operating system provides six resource classes to an application: persistent storage (disk), user interface, network, randomness, system time, and memory. Peripheral devices such as printers, USB devices, etc, are abstracted by the OS into file objects and are thus part of the persistent storage category.

While it is possible to provide routing rules for all six resources, we have found that this is unnecessary. An application may choose to forward system requests to the commodity OS for two reasons: either it wants to use the resource as a communication channel with another application, or it does not need the resource to be trusted and thus wishes to include the resource outside of its TCB. As a result, persistent storage, user interface and the network are routed by Proxos because these are resources that applications either use to communicate, or may not need to trust. System time and randomness are never routed because they cannot be used as communication channels, and are provided by the underlying VMM without increasing the application's TCB. Finally, memory related

```
# Rules Section
# route accesses to /etc/secrets to private OS
DISK:("/etc/secrets", priv_fs)
# route accesses to UNIX domain socket bound
# to /tmp/socket and TCP socket bound to peer
# 192.100.0.4 port 1337 to private OS
NETWORK:("unix:/tmp/socket", priv_unix),
        ("tcp:192.100.0.4:1337", priv_tcp)
# route all accesses to stdin, stdout
# and stderr to private OS
UI: (*,priv_ui)

# Methods Section
# individual methods in the private OS
# that are bound to system calls
priv_fs = {
  .open = priv_open,
  .close = priv_close,
  .read = priv_read,
  .write = priv_write,
  .lseek = priv_lseek
}
```

Figure 3.2: Routing Example. This example shows a simple set of routing rules that protects operations on a particular file, two network sockets, and the standard I/O streams.

system calls (such as `brk` and `mprotect`) are used to indirectly manipulate page table entries. However, a private application would never trust a commodity OS with control of its page tables since this would imply granting the commodity OS access to the private application's memory. Therefore, it does not make sense to route memory-related system calls. All non-routable system calls are directed to functions provided by Proxos.

Based on this model of operating system resources, we have designed a simple language that allows the application developer to specify which system calls will be routed to the commodity OS, and which to the private OS. Figure 3.2 shows a stripped-down example of a routing specification in our language. Lines prefixed with a "`#`" are comments. The Rules section consists of three declarations, one for each of the routable resource classes. The specifications for the disk and network resource classes are a list of tuples, where each tuple describes the particular resource, and a table of function pointers used to access the resource. In this case, the specification for the user interface (`UI`) has "`*`" as a resource

description because the application wants to route all three standard I/O streams (i.e. `stdin`, `stdout`, and `stderr`) to the private OS. The example also specifies that access to any file with name `/etc/secrets` should be handled by methods in the private OS. The same is true for system calls to any UNIX domain socket bound to `/tmp/socket` and to any TCP socket with a peer IP address of 192.100.0.4 on port 1337. By default, Proxos will route all system calls to resources that do not match any rule to the commodity OS.

The Methods section defines which methods in the private OS will handle system calls from the application. When the application attempts to open the file `/etc/secrets`, Proxos will call the `priv_open` method in the private OS to handle the request and return a file descriptor. All subsequent system call operations (such as `close`, `read`, `write` and `lseek`) on the file descriptor associated with that file will also be forwarded to the associated private OS method in the table. On the other hand, any system call on the file that is not in `priv_fs` will be forwarded to the commodity OS. Method tables for `priv_ui`, `priv_unix` and `priv_tcp` are not shown in the figure, but must also be specified by the application developer.

Rather than specifying trust policies by partitioning code, or by restricting abilities, specifying policies by partitioning interfaces to resources results in a more compact and intuitive policy description. Further, our specification language allows the application developer to use the same names for resources as those in the source code, making the routing rules easier to write and understand.

# Chapter 4

# Implementation

There were several requirements that dictated which underlying system we chose to implement our Proxos prototype on. First, we needed a way of "hoisting" a commodity OS to a lower privilege level and inserting our own privileged code beneath it. Second, the system had to provide isolation between the private applications and the commodity OS, but at the same time allow some controlled communication between them. In light of these requirements, we eventually settled on using the Xen VMM [3] and Linux as our commodity OS for our experimental substrate. However, we believe that the features required by our system could be provided by any VMM or microkernel.

In this chapter, we describe the three main components we implemented in building our prototype. First, we describe our modifications to Xen and Linux to provide support for starting private applications, and to forward system calls between VMs. Second, we describe our Proxos operating system proxy prototype, which routes system calls to either the commodity Linux kernel or to private OS methods. Finally, we describe some private OS methods that we have implemented.

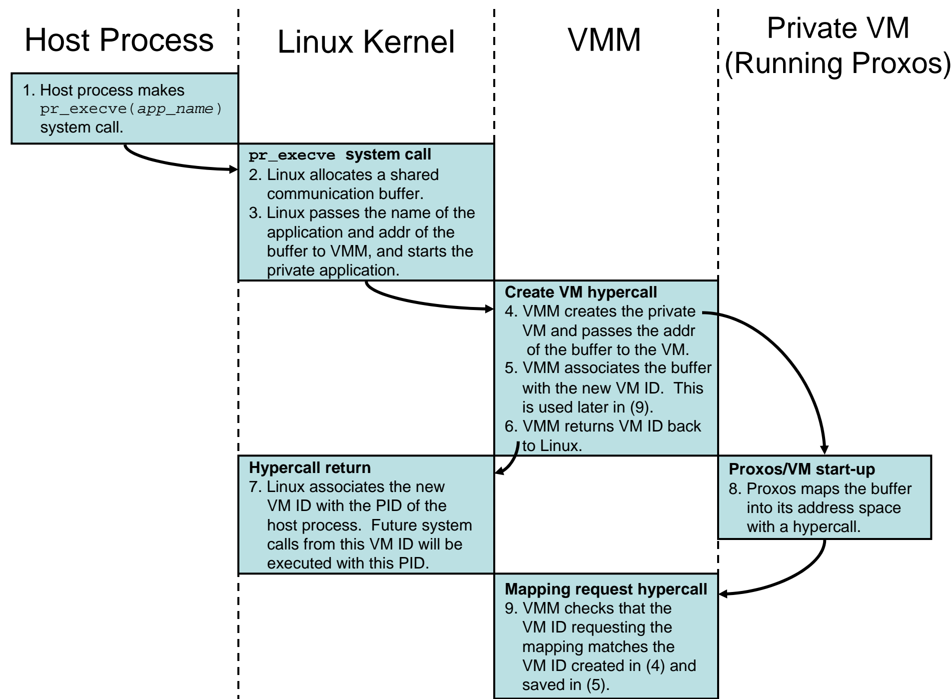


Figure 4.1: Private Application Start-up Sequence. The steps are arranged into columns with the titles at the top indicating what system component each step takes place in.

## 4.1 Modifications to the VMM and the Commodity OS

Modifications made to Xen and the Linux kernel can be categorized into three components: the start-up and shutdown of private applications, a facility for forwarding system calls between VMs, and a trusted path facility.

Since the Linux kernel and private applications do not trust each other, the private application start-up process must make several guarantees. First, the private application must not be able to gain any privileges beyond those of its host process. This implies that the Linux kernel must always be able to attribute system calls routed to it to the host process that initiated the private application forwarding the system call. Second, a compromised commodity OS should not be able to initialize a private application in

an unsafe state. Finally, the private application should not be able to access any Linux kernel memory that the kernel has not authorized it to.

The VMM administrator registers private applications with the VMM via a configuration file. This file assigns a name to each private application and sets start-up parameters for each private VM. Later, when the host process starts a private application, it will use this name to indicate to the VMM which private application to start. Figure 4.1 describes the private application start-up process used in our prototype in detail.

In Step 1, private applications are started using the `pr_execve` system call that we added to the Linux kernel. `pr_execve` is the private application analog to the `execve` system call and, like `execve`, takes the name of the private application to be started as its argument. `pr_execve` causes the current process to become the host process for the private application.

In Step 2, the Linux kernel allocates a shared buffer that is used later for system call arguments forwarded to it from the private application. The kernel passes the address of this buffer to the VMM in Step 3, and at the same time signals the administrative VM to start a new VM for the private application with a hypercall we introduced. The administrative VM will only start the private VM with parameters set by the system administrator, ensuring that even a compromised Linux OS can only start private applications from a known, safe state. Note that a compromised Linux OS may start a private application different from the one the host process requested and attempt to get the user to use the wrong private application. To detect this, Proxos relies on application level safeguards such as the trusted path used in our web browser or cryptographic keys used in our SSH private server. We will discuss both of these applications in chapter 5.

In Steps 4 to 7, the VMM creates a new private VM for the application, informs the Proxos in the private VM of the location of the shared communication buffer, and informs the Linux OS of the identity of the new VM (by giving it a VM ID). Then, in Step 8, Proxos tries to map the shared buffer into its address space via another hypercall.

Originally a privileged hypercall, we modified this hypercall so that private VMs may use it. However, we also added an extra check to ensure that the VM making the mapping request in Step 9 is the same as the one to which the VMM originally passed the shared buffer address in Step 4.

Private application shutdown is much simpler as there are no security guarantees to be made. If the private application initiates the shutdown, then it informs the VMM via a standard hypercall. We extended this hypercall to notify the Linux kernel, so that the kernel may terminate the host process accordingly. Even if the private application has not terminated, the kernel may still forcibly destroy the host process (by killing the process). However, the kernel does not have the privileges to force the private application to shutdown, so by killing the host process, the kernel can only revoke the private application's ability to access commodity OS resources.

Another set of modifications allow private applications to forward system calls to the Linux OS. The goal is to reduce the latency of forwarded system calls by reducing the number of domain crossings. Xen already provides a facility that allows VMs to send events to each other. By combining this with the shared buffer between the Linux OS and the private application, we were able to add a simple RPC mechanism to Xen. We then made modifications to the Linux kernel to allow it to efficiently execute forwarded system calls. When the Linux kernel receives the system call arguments, it determines the appropriate host process to wake up by examining the source of the RPC and comparing that to information it recorded in Step 7 of the start-up sequence. As the host process is about to be scheduled, a trip into user-space can be saved by placing the system call arguments in the appropriate registers and transferring control directly to the system call handler in the kernel. When the system call handler completes, the kernel sends the return value back to the private application via an RPC response message and returns the host process to the queue it was in before the forwarded system call arrived. As a result, our prototype handles forwarded system calls without any domain crossings in

the Linux OS.

Finally, we also needed the VMM to provide a trusted path facility so that private applications can communicate directly with the user without having to trust the commodity Linux OS. This would prevent a compromised Linux OS from masquerading as a private application, as well as prevent a compromised Linux OS from eavesdropping on communication between a user and a private application. To support this, the VMM provides user interface facilities such as a console driver and graphical window system. If the private application wants to use these facilities, it routes system calls on standard I/O streams (i.e. `stdin`, `stdout` and `stderr`) to private OS methods, which will forward the requests to the VMM console driver. Similarly, it routes X window operations to private OS methods that will translate them into the appropriate operations on the VMM window system. The implementation of minimal trusted window systems on secure kernelized systems has been studied in the literature [11, 34]. Rather than re-implement these in our prototype, we simply provided an emulation of their functionality, but do not make any effort to reduce the amount of code that is added to the VMM. We did this by running an X server on Xen's administrative VM and using nested X servers to give each VM its own separate X interface.

We found that modifying Xen and Linux to allow private application start-up and shutdown, as well as forwarded system calls, had very little impact on the size of the Xen TCB. Many of the facilities needed were already present in the Xen VMM and we only had to make these accessible to unprivileged VMs and add checks to make sure they could not be abused. The only component that increases the code base of the VMM significantly is the graphical user interface. A significant portion of this component can be implemented outside of the trusted computing base of the VMM [11, 34], but exploring the design of trusted window systems was not a goal of our prototype.

## 4.2 The Proxos Prototype

Our prototype is derived from the *Minimal OS* example that comes with the Xen 2.0 source code. Proxos runs in a single address space and supports only one private application. Our current implementation is also single-threaded, although we plan to support threads in the future. Apart from providing basic memory and page table management, Proxos also contains: a block driver that supports raw accesses to a private block device exported by the VMM; and a console driver that provides direct access to the Xen console. Our prototype does not provide a TCP/IP stack or a network driver. We found these unnecessary as many security-sensitive applications already assume the network is not trustworthy and employ cryptographic safeguards such as SSL to protect network communications. This allows us to safely reuse the network services of the commodity OS.

Proxos uses operating system abstractions to determine where to route system calls at run time. In the case of Linux, the abstraction used by applications to access resources is a file descriptor. Initially a file descriptor is bound to a resource via a system call such as `open` or `socket`. Subsequent operations on that resource are then performed by naming the descriptor in the system call.

The design of Proxos is very simple, and is similar to the way virtual file system methods are implemented in Linux. Routing rules for the application are converted into lookup tables, which are then compiled into the Proxos library and linked with the private application. When descriptors are created, Proxos compares the name of the resource they are being bound to with the routing rules specified for the application. For example, if a file descriptor is being created via an `open` system call to a file, Proxos compares the name of the file being opened with the list of tuples provided in the `DISK` resource class. If a match is found, Proxos uses methods from the method table specified in the matching routing rule to handle subsequent system calls on the descriptor. Proxos provides a set of default methods which route untrusted system calls to the Linux OS. If a routing rule

specifies a private OS method to be called, Proxos transfers control to the appropriate location in the private OS.

The private application uses file descriptors to name objects in both the private OS and the commodity OS. File descriptors in the commodity OS are allocated from a name space independent of the one the private application is using. Upon opening a new file in the commodity OS, Proxos may find that the commodity OS has assigned a file descriptor number that the private application is already using to name another object in the private OS. As a result, Proxos translates between the file descriptors used by the private application, and those used in the commodity and private OSs.

Most routable system calls can be routed transparently to the Linux OS. However, the `fork`, `execve` and `select` system calls have slightly different semantics. When forwarded, the `fork` system call will cause the host process in the Linux OS to fork. The forwarded `fork` creates concurrency on the Linux OS side, but the application in the private VM will still contain only a single thread of execution, so parent and child code must be executed sequentially. After the `fork`, the private application specifies whether system calls it forwards to the Linux OS should be executed by the parent process or the child process. This is done by setting the *target PID flag* in Proxos to indicate the process ID (PID) of the process that should be the recipient of system calls forwarded to the Linux OS. The value of this flag accompanies every system call Proxos forwards to the Linux OS. The Linux OS checks that the PID specified by the flag belongs to either the host process, or a child of the host process. These semantics imply that forwarding `fork` system calls requires the developer to make any concurrent code sequential in the private application. To support standard `fork` semantics, the underlying VMM needs to be capable of duplicating the address space of the private application (preferably using copy-on-write for efficiency). While we did not support this in our prototype, we note that others have proposed adding such functionality to Xen [37].

The semantics of forwarded `execve` system calls are also slightly different. If the

`execve` system call is made without a fork, the host process will terminate and a new program will take its place. If the new process is not willing to host system calls forwarded to it, the private application will be unable to forward system calls to the Linux OS. More commonly, a recently forked process will execute `execve`. In this case, the private application will lose the ability to forward system calls to the child, but retain the parent as the host process. More details on how `fork` and `execve` are used in private applications will be given in the description of our port of the SSH server in Section 5.2.

Finally, `select` has a slightly different behavior under Proxos than its Linux counterpart. `select` allows applications to listen on several file descriptors simultaneously and notifies them when there is activity on any of the descriptors. In Proxos, an application may execute a single `select` on file descriptors from both the commodity OS and the private OS. However, Proxos forwards system calls by making *synchronous* inter-VM RPCs. This limitation of our current prototype prevents Proxos from routing `select` system calls to both OSs simultaneously, so it serializes them and imposes a time-out on each `select` call. Proxos will alternate between which OS to execute `select` on first to ensure no file descriptor is starved. The `poll` system call has the same behavior as `select` in our system. The consequence of this is that events on file descriptors that happen close together may not be delivered to the private application in the same order that they occurred because Proxos may be polling the other OS instance when the first event occurs. However, we have not seen this to be an issue and, to the best of our knowledge, Linux makes no such ordering guarantees either.

## 4.3 Private OS Methods

In our prototype, we have implemented two example private OS components: one that implements a private file system, and one that implements a trusted path by forwarding standard I/O streams and X window messages to the VMM.

A private file system allows the private application access to persistent storage that is protected from tampering by the Linux OS. We wanted to implement this by adding as little code to the private VM as possible, as any code we add increases the TCB of the application. Rather than implement an entire file system, our private file system outsources most of its functionality to the commodity Linux OS through forwarded system calls, but maintains the secrecy of any information stored by encrypting all data before writing it to the Linux file system [18]. To protect the data from tampering and replay, hashes of all files stored on Linux by the private file system are kept on a private block device available directly from the underlying VMM. Doing this significantly simplifies the file system implementation, as all that is needed are the cryptographic functions, some code to manage file system buffers, and block device drivers to store the file system hashes. The drawback is that a compromised Linux OS could potentially deny the private application access to files that the private file system has saved. However, our applications typically depend on the Linux OS for other services as well, so no forward progress guarantees are broken by this.

In our prototype, the private OS implements a trusted path by routing operations on standard I/O streams and the X server's socket to the VMM. The private OS methods translate system calls on standard I/O streams to operations on Xen's console driver and route system calls on the X server to the administrative VM. A host process on the administrative VM then executes the routed system calls on a socket connected to a nested X server instance that is separate from the one that the commodity Linux OS is using.

## 4.4 Discussion

With the exception of modifications to the Linux kernel, all components implemented in our prototype will be part of the application TCB. As a result, we placed a lot of emphasis

| Component | Lines of Code |
|---|---|
| VMM modifications | 656 |
| Linux modifications | 4380 |
| Proxos | 7348 |
| Private File System | 1817 |
| Trusted Path | 1313 |

Table 4.1: Number of lines of code in each component in our Proxos prototype. The VMM modifications do not include the X server running in the administrative VM.

on keeping the impact on code size and complexity small, especially with respect to the VMM. One caveat is that Proxos does not need to support every system call that Linux exports. For example, Proxos does not support system administration calls, such as those to control swap devices, or load and unload kernel modules, as private applications will not need to make such requests. Out of the 289 system calls of the Linux 2.6.10 kernel, our Proxos prototype only needs to support (either internally or by forwarding) 56 of them to run most applications. However, we fully expect this proportion to increase as Proxos matures. The size of the components in our prototype are given in Table 4.1.

# Chapter 5

# Applications

In this chapter, we describe three applications that we have ported to Proxos. We selected applications that will benefit from partially trusting a commodity OS, and illustrate interesting issues that arose when porting them. Our first application is a secure web browser that protects user information. Our second application is an SSH server that protects system-critical information such as passwords and host keys even if the commodity OS is compromised, but still allows users who login to gain a full shell on the commodity Linux OS. Our final application is an SSL certificate service that we use with an Apache web server to implement SSL transactions. In this case, the private keys corresponding to the certificate are protected.

## 5.1 Secure Web Browser

A serious threat to the security and privacy of users is spyware, which is malicious software that is surreptitiously installed on machines and monitors the web surfing habits of users. While the goal of most spyware is to collect usage data for marketing, spyware has been shown to decrease the security of user system by recording and transmitting confidential information that it has collected [27, 32].

Spyware collects information by either monitoring the user's keystrokes, or by scraping

files where web browsers have recorded user information. We ported Dillo [8], a simple graphical web browser, to Proxos and configured the routing rules so that all user I/O is directed to the trusted VMM user interface, thus creating a trusted path, and all sensitive data that Dillo saves to disk is directed to our private file system. No other rules are specified, and thus other network operations such as HTTP requests are routed to the Linux OS by default (for extra security, the user should use HTTPS to encrypt traffic between the browser and the web server to prevent any spyware on the Linux OS from observing or tampering with it). Similarly, any documents or executables that the user downloads from the Internet are saved to the Linux file system. In addition, any external helper applications that Dillo invokes will be transparently created and executed on the Linux OS.

For the most part, no source code modifications were required to port Dillo. The only necessary modifications were due to Dillo's use of graphical themes, which are implemented as code that is dynamically loaded at run time based on the theme the user selects. In our prototype, it is not safe to load code from the Linux OS, since an adversary may have tampered with it. To support the default theme, we removed the code that loads themes at run time and statically linked the default theme into the Dillo private application. In theory, code could be safely loaded from the Linux OS if encrypted and accompanied by a valid signature that the private application could verify, but our current prototype does not support this.

## 5.2 SSH Authentication Server

Often when attackers compromise a system, the system administrator is not only forced to rebuild the entire system from scratch to ensure that any malicious software has been removed, but also to perform the arduous task of tracking down every user and ensuring that they change their passwords in case the attacker has been able to learn some of

the old passwords. Similarly, she must change any cryptographic host keys, which the machine uses to authenticate itself, and distribute new keys to all parties that the machine interacts with. Being able to ensure the secrecy of user passwords and the host keys of a system after a security compromise would save the administrator significant time and effort.

To demonstrate the utility of Proxos in protecting the secrecy of sensitive data, we ported the OpenSSH authentication server (version 3.9p1) to Proxos. The OpenSSH server accesses several sensitive resources including configuration files, the password file and the host key file. We wrote routing rules to store the password file, host key, and global configuration files on the private file system. The SSH server also performs network operations, but no rules are specified for `NETWORK` resources since OpenSSH is designed to function with an untrusted network. Other than the routing rules, only two modifications involving the `fork` system call were required to implement a private SSH server application. The architecture of our new private SSH server is shown alongside the architecture of the original SSH server in Figure 5.1. One modification arose because the SSH server requires some concurrency to allow multiple users to authenticate simultaneously. The native version of SSH handles this by having a parent process listen on the SSH port, and then spawning a child for every connection the parent receives. Our private SSH server still has the listening parent as a native Linux application, but implements the children as private applications. When the listening parent detects a new connection, it forks a child (on the Linux OS), which then uses `pr_execve` to instantiate a private SSH server VM, and in doing so becomes the host process for the new VM.

The private SSH server starts-up and reads the sensitive data from the private file system, and then proceeds with user authentication. If a user logs in using private key authentication, the private SSH server will need to access the public keys the user has placed in a file in their home directory on the Linux OS. Proxos provides access to the user's keys without any extra configuration – since the user's key files do not match any

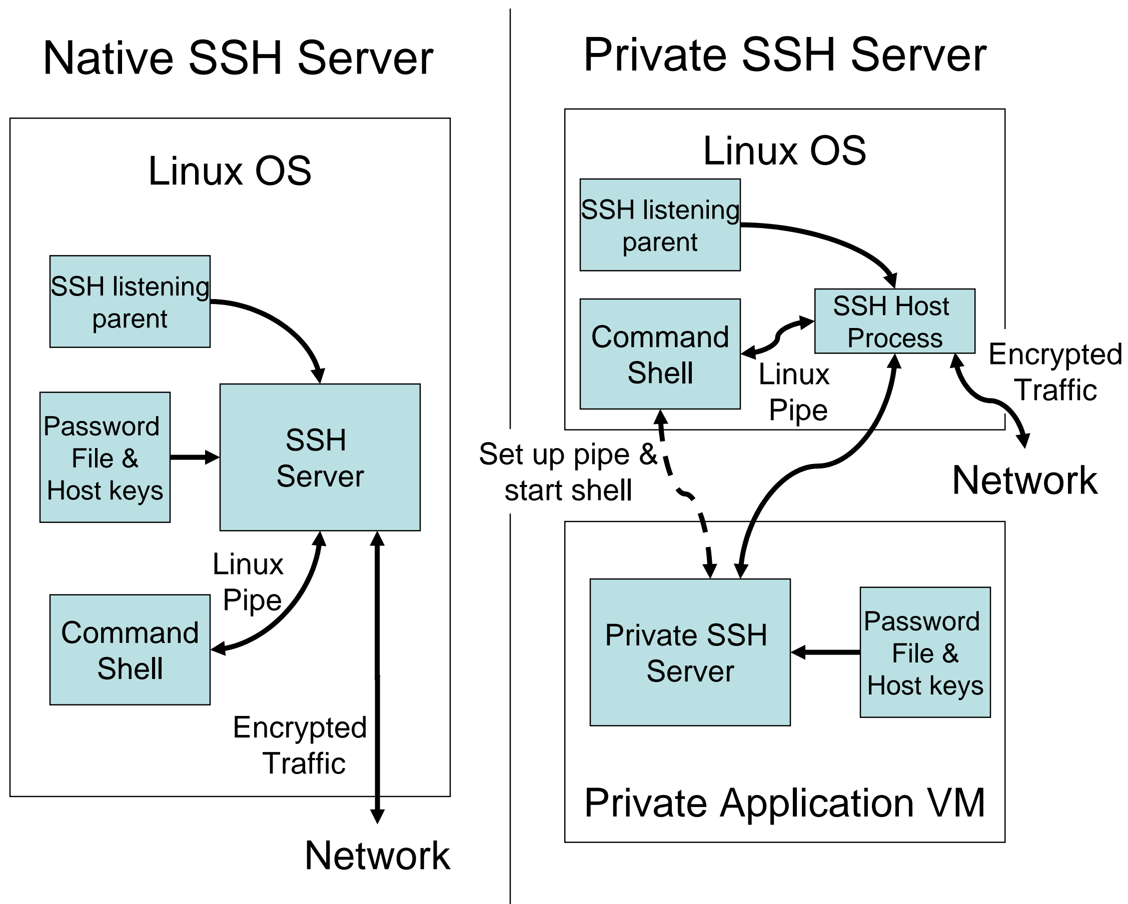


Figure 5.1: Comparison of the original SSH server and our private SSH server.

routing rules, requests to them will be forwarded to the Linux OS by default. If the user authenticates successfully, the native SSH server forks a child that will execute a command shell. Before the child starts the command shell, the native SSH server creates a pipe between itself and the command shell redirecting all input and output from the shell to itself, so that it can encrypt any shell output before sending it to the network, and decrypt any shell input coming from the network. In our version, the private SSH server changes the Proxos target PID flag to point to the new child after the fork, and then executes the child code, forwarding the system calls required to set up the pipe and start the command shell. After this, it changes the target PID flag back to the parent and executes the SSH server code. The shell will pipe all input and output through the

host process to the private SSH server, which encrypts and decrypts data as appropriate between the shell and the network.

## 5.3 SSL Certificate Service and Apache

Next, we explored the performance impact of Proxos on Apache with SSL. As in the SSH server, the Apache server relies on concurrency so we only ported the `crypto` library portion of the OpenSSL library to Proxos, and left the Apache web server on the Linux OS. The `crypto` library uses confidential private keys stored in the SSL certificate, which would be protected if the web server was compromised. Our port uses Apache version 2.0.52 and version 0.9.7g of the OpenSSL library.

To setup SSL sessions, Apache makes calls to the OpenSSL library, which uses the OpenSSL "engine" interface to invoke the `crypto` library. We modified the engine interface to spawn a private application that will use the private key in the server's SSL certificate to sign challenges during an SSL handshake. Unfortunately, this operation is called on every HTTP request that uses SSL (i.e. an HTTPS request), and would give very poor performance because each request results in the instantiation and shutdown of a private VM. To remove the frequent instantiation and shutdown of the private VM, we modified Apache to spawn a process when it starts-up, which will act as the host process for a single private SSL certificate application. Apache was also modified so that a portion of the shared buffer between the host process and the private application is mapped into the address space of each Apache thread. To process an HTTPS transaction, a thread enqueues the signing request on the shared buffer, sends a signal to Proxos for processing and sleeps until the request has been processed. Since multiple Apache threads will be accessing the shared buffer, we also added the appropriate synchronization between the threads to prevent races.

| Application | Rules | LOC Modified |
|---|---|---|
| Dillo | 53 | 22 |
| SSH Server | 35 | 108 |
| Apache & OpenSSL | 28 | 667 |
| Glibc | | 218 |

Table 5.1: Size of routing rules and number of LOC modified for each application.

## 5.4 Discussion

The size of our routing rule descriptions, along with the lines of code that were modified for each of the applications, as well as Glibc (version 2.3.3), is given in Table 5.1. In porting these applications we found that what often takes some time are modifications to application source code that are required to support operations like `fork` and `execve` in the private SSH server, or to statically link in dynamic code in Dillo. Apache required more effort since several threads could make challenge-signing requests simultaneously, and this required careful arbitration and synchronization to preserve performance. We find these results encouraging – Proxos enables the application developer to remove the entire commodity OS kernel and privileged applications from the TCB of the private application by modifying on the order of several hundred lines of code in the application, and writing around 50 lines of routing rules.

# Chapter 6

# Evaluation

The performance of VMMs versus native operating systems has been well studied in the literature [3, 4]. To ascertain the overhead introduced by Proxos, we compare the performance of our system against a system running an unmodified Linux kernel executing on an unmodified Xen VMM. We first use microbenchmarks to better understand the components that contribute to the cost of making forwarded system calls from a private application. Then we evaluate the performance of the SSH and Apache/SSL Certificate applications described in chapter 5 on our system. All tests were performed on a machine with a 3GHz Intel Pentium 4 processor, 1GB of RAM, a 7200 RPM Serial-ATA disk with 8.9 ms seek time, and a 100Mb Ethernet NIC. Our prototype is built on Xen 2.0, with Fedora Core 3 Linux running a 2.6.10 kernel as the commodity OS, and its performance is compared against vanilla versions of the same software. For our runs, 768MB of RAM were allocated to the commodity OS, and the rest was used for Xen, the administrative VM, and private applications.

## 6.1 Microbenchmarks

To analyze the overhead of a system call forwarded from the private application to the Linux OS, we must first understand the individual components that make up a forwarded

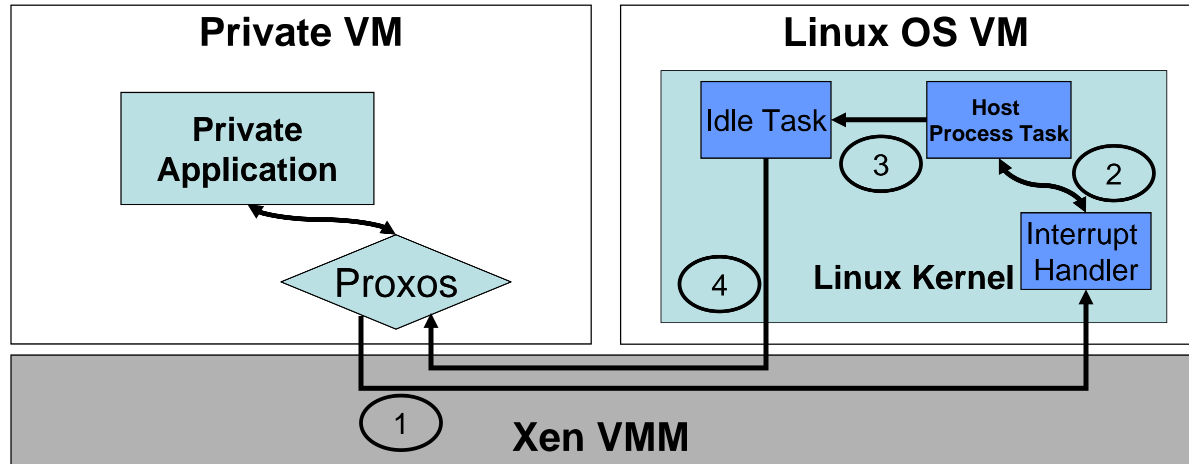


Figure 6.1: Breakdown of costs incurred in a forwarded system call.

system call. These costs are illustrated in Figure 6.1. In Step 1, Proxos sends an event to the Linux kernel, notifying it of the forwarded system call, and then yields the processor, causing a VMM context switch into the Linux OS VM. In the Linux kernel, a virtual interrupt handler receives the event and enqueues the system call request on the process descriptor of the host process. In Step 2, we wait until the host process is scheduled. On a lightly loaded system, this incurs only the cost of another context switch within the Linux kernel, but may take more time if the Linux kernel is heavily loaded. After the Linux kernel executes the system call, it will not yield the processor back to the VMM until either the VMM scheduler decides to preempt the Linux OS VM, or the kernel runs out of runnable processes and schedules the idle task in Step 3. Finally, in Step 4, another VM context switch occurs and Proxos can receive the result of the system call. While this accounts for four context switches, there is actually a fifth context switch because Xen will schedule the administrative VM in either Step 1 or Step 4.

We ran the system call latency benchmarks in the LMbench 2.5 microbenchmark suite [26] in a private VM configured to forward all system calls to an idle Linux OS VM, and summarize our results in Table 6.1. We also used the context switch microbenchmark in LMbench and measured the minimum cost of a context switch to be 2.88$\mu$s on our

| **Benchmark** | **Linux** | **Proxos** | **Overhead** |
|---|---|---|---|
| NULL system call | 0.37 | 12.88 | 12.51 |
| fstat | 0.57 | 14.28 | 13.71 |
| stat | 8.76 | 25.98 | 17.22 |
| open & close | 14.57 | 47.18 | 32.61 |
| read | 0.45 | 13.51 | 13.06 |
| write | 0.42 | 13.24 | 12.82 |

Table 6.1: Forwarded system call latencies on LMbench microbenchmarks. All measurements are given in $\mu$s.

machine. As a result, the expected five context switches would take approximately 14$\mu$s, which tracks well with the measured results. This cost is added to every system call except for `stat` and `open`, whose larger overhead can be explained by the fact that each context switch changes virtual to physical page mappings, and causes a TLB flush. Since both `stat` and `open` take a filename as an argument, the Linux kernel must make several queries to the buffer cache to find the correct inode (LMbench ensures that the inodes required to access the files are cached in memory), which will result in TLB misses. These misses do not occur when the benchmarks are run directly on Linux because the kernel never switches to another process, so no context switches occur.

## 6.2 Application Benchmarks

We now evaluate the overhead imposed on our private SSH server and SSL certificate service. Like our microbenchmarks, applications incur overhead when system calls are routed to the commodity OS. To evaluate the average overhead a forwarded system call experiences, we used an SSH client to login to our private SSH server over the loopback device and measured the time taken to copy files ranging from 32MB to 256MB over the

SSH connection. Each file transfer was performed five times on both the private SSH server and a native SSH server running on Xen. The standard deviation was less than 1.5% across our measurements. Figure 6.2 plots the average difference in time taken by the private SSH server over the native SSH server to transfer a file, against the number of forwarded system calls the private SSH server made. We perform linear regression on the average values and found a correlation of 0.92, indicating that the overhead is well correlated with the number of system calls. We then estimate the start-up component to be 0.72s and the per-system call cost to be 15.7$\mu$s. For large files, where the cost of start-up has been amortized, the private SSH server only takes 6.0% longer to transfer the same file as the native SSH server. Note that since this overhead is comparable to the variance in our measurements, the estimated system call overhead should not be taken too literally, and is merely a rough approximation.

We suspected that a large part of the start-up cost for the private SSH server is due to VM creation. We confirmed this by measuring the time to start an empty private VM, which is approximately 0.35s. Starting a Xen VM requires the use of several user-space scripts in the administrative VM, making it very expensive, and we have not made any effort to optimize this operation. The remaining 0.37s is the time the private SSH server uses for initialization, which includes the time it takes to read in sensitive data from the private file system. This operation requires several cryptographic operations to decrypt the data and verify the authenticity of files stored on the commodity OS file system.

To evaluate the performance impact of our private SSL certificate application, we used Mindcraft's Webstone benchmark [39] extended with SSL. We configured the benchmark with 150 SSL clients, which was enough to fully load an Apache server on Xen. The same number of clients was used to measure the amount of bandwidth our Proxos-enabled web server could support. We expected the Proxos-enabled web server to introduce low overhead because HTTPS transactions mainly perform computation and make very few system calls. Our experiments show that there is actually a slight increase in throughput

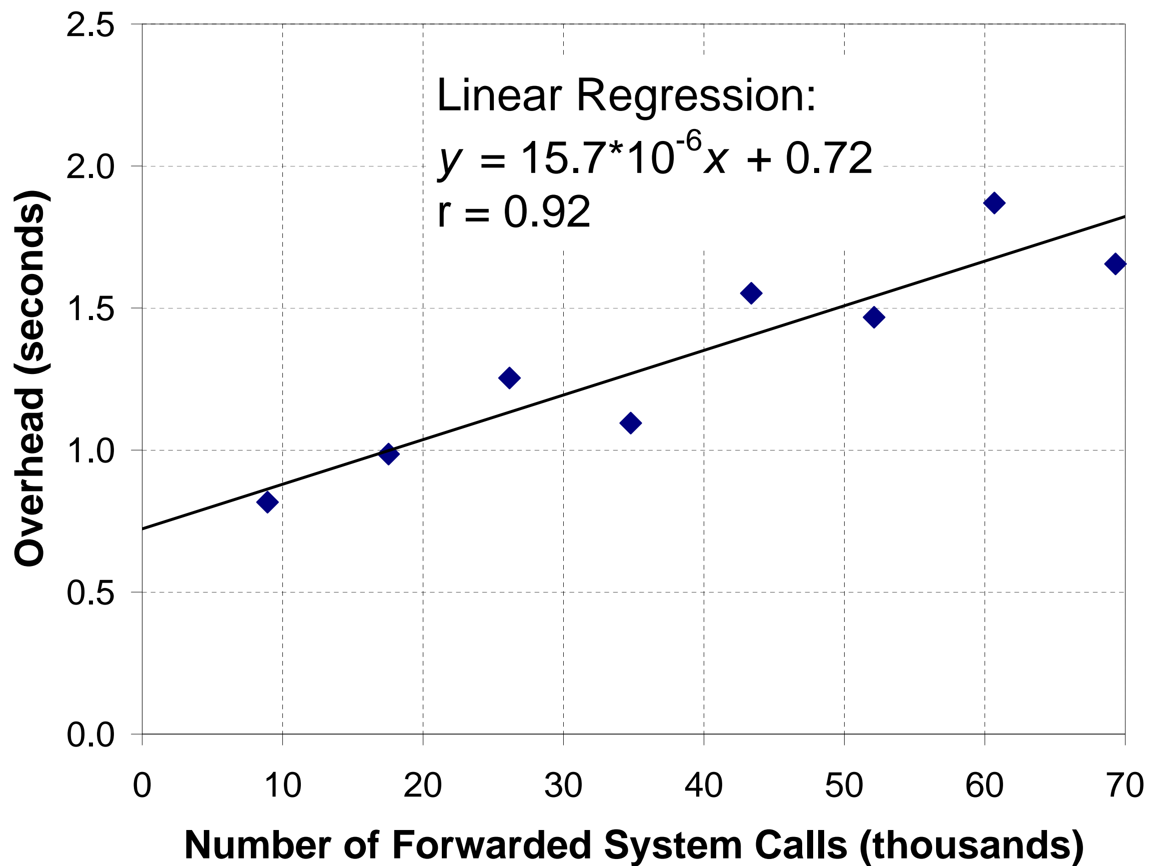


Figure 6.2: SSH benchmark. We plot the overhead of the private SSH server versus the number of system calls forwarded to Linux.

– 5.04Mb/s for the Proxos-enabled web server as compared to the native web server's throughput of 4.75Mb/s. From this, we surmise that the overhead Proxos introduces is very low and that the changes we made porting the system likely perturbed the system in such a way as to produce a slight performance gain.

# Chapter 7

# Related work

We compare our work with other research projects along three axes. We begin by comparing our system with projects that provide applications with some degree of flexibility in customizing their operating system. Next, we look at projects that enforce access control policies to improve the security of operating systems. Finally, we explore recent research projects that leverage the isolation properties of a VMM or a microkernel to isolate security-sensitive applications.

## 7.1 Systems with flexible architectures

Traditional operating systems export generic abstractions that are used to support a broad class of applications. However, by restricting applications to a generic interface, traditional operating systems limit the functionality, performance and simplicity of applications. Exokernel [10] and Denali [40] provide a flexible architecture that allows applications to customize the underlying system to meet their specific requirements. Instead of exporting traditional OS abstractions, the Exokernel system takes the unconventional approach of directly exporting the low-level interface of the hardware to higher-level applications. Similarly, Denali virtualizes the hardware and exports to higher-level applications a simplified VM abstraction that enhances performance and simplicity. An

application running on those systems implements its own *LibraryOS* which contains services such as virtual memory, scheduling, IPC and filesystem that have been tailored to the application's requirements. Exokernel and Denali are similar in structure to Proxos as they both allow applications to customize and implement their own library of system services. In Exokernel and Denali, applications customize their *LibraryOS* for performance and functionality. In contrast, on Proxos, applications customize their *LibraryOS* to provide for trusted services. As for services that do not need to be trusted, applications can exploit Proxos's system call routing facilities to reuse the services found in the commodity OS. Proxos also differs from these systems is in its objective. While the goal of the former systems is to give applications some ability to customize the underlying OS for performance and functionality, Proxos gives applications the ability to customize the trust relationship between applications and the OS kernel.

## 7.2 Operating system security

Several projects try to enhance the security of operating systems by enforcing the principle of least privilege [31] where processes are assigned with the minimum set of privileges required to perform their job. Eros [33] and KeyKOS [29] are capability-based operating systems that enforce the principle of least privilege. In a capability-based system, all objects are accessed by means of a capability, which is a token that gives its holder the ability to access a particular resource. In Eros and KeyKOS, processes are given the minimum set of capabilities needed to perform their task. The capabilities are unforgeable and tamper-proof. All transfer of capabilities among processes is mediated by a reference monitor that imposes access control rules. By restricting processes to their minimum set of capabilities, capability-based systems limit the damage that can be done by a compromised process.

Asbestos [9] is a new OS designed to control the flow of information in the sys-

tem. In Asbestos, security labels are assigned to processes. Using the labels, Asbestos can specify the interaction among processes and control the flow of information to enforce mandatory access control (MAC) policies. To support high-performance services, Asbestos introduces an *event process* abstraction that allows a single-process server to service several concurrent users while still isolating the private data and states of the users. Asbestos's *event process* enhances the security of services by containing the effects of an exploit to affect only one user's data.

While the above systems provide useful mechanisms to contain the damage of an exploit, they are all based on architectures that differ significantly from traditional commodity OS. Applications wishing to harness the security properties of those systems have to be rewritten for the new architectures. In contrast, Proxos retains the same application interface as a commodity OS and security-sensitive applications wishing to take advantage of Proxos do not require extensive porting. Further, applications that do not require the security advantages of Proxos can remain in the commodity OS and suffer no overhead.

SELinux [25] implements MAC mechanisms in the Linux kernel that allow for the enforcement of fine-grained MAC policies. With SELinux, each process is associated with a domain and each object is associated with a type. A trusted security administrator configures the system with a set of rules that dictates the allowable access of domains to types and the allowable interaction between domains. SELinux does not suffer from the shortcomings of traditional Linux systems that rely on setuid/setgid programs to access privileged resources. By confining processes to domains, SELinux can restrict processes to the minimum privileges necessary to do their job and hence contain the damage caused by flawed or malicious programs.

The fine-grained access control offered by SELinux has its costs. SELinux policies are large and complex – the size of the default policy set for the Fedora Core 3 Linux distribution has over 290,000 rules and consumes more than 7MB of kernel memory. In

contrast, Proxos's interface routing configurations are typically around 50 lines long or less. Aside from the difference in size, the configuration of SELinux and Proxos carries a more subtle difference. The security of an SELinux system is dependent upon the administrator to correctly configure all applications to execute with their least privileges. On the other hand, to protect a security-critical application, Proxos only requires that the rules for that one application to be configured correctly. The security of the private application running on Proxos remains unaffected by the configuration state of all other applications running on the untrusted commodity OS. Finally, SELinux also differs from Proxos in that the former system still depends on the correctness of the kernel, while the latter maintains the integrity and confidentiality of protected objects even in the face of a full compromise of the Linux kernel.

## 7.3 Systems that provide isolation

Modern commodity OSes are becoming overly complex and typically run a myriad of applications that are poorly isolated from one another. As a result, the compromise of any one of the applications usually leads to a full system compromise. Several research projects protect security-sensitive applications by isolating them from the rest of the potentially untrusted system.

Terra [13] leverages the isolation properties of a VMM to isolate applications with differing security needs. Terra exports two types of VMs: an *open-box* VM used to run a general purpose commodity OS together with its typical set of applications; and a *closed-box* VM used to run security-sensitive applications where the software stack can be tailored to the security requirements of the applications. The isolation maintained by Terra is coarse-grained as applications can either execute in the *open-box* VM or be completely isolated in a *closed-box* VM. This coarse-grained isolation may not be suitable in scenarios where the application running in the *closed-box* needs to use the services or

interact with other applications running in the *open-box*. In such situations, Terra would need to include those services and other applications in the *closed-box*. This leads to a duplication of resources across VMs, an increase in the complexity and a reduction in the security assurance of the software running in the *closed-box*. Proxos differs from Terra as it allows security-critical applications running in a private VM to reuse the resources of the commodity OS in a controlled fashion.

Other projects use a kernelized system approach to isolate applications. Perseus [28], uses the Fiasco [17] microkernel to run a commodity operating system and security-critical applications in separate protection domains. Perseus relies on the underlying microkernel to restrict IPC and to isolate the protection domains. The Nizza project [35, 16], runs the L4Linux [15] commodity OS on the Fiasco microkernel. Nizza goes one step further in securing Linux applications. Nizza requires a developer to analyze the source code and to manually extract the security-sensitive components of the applications into separate *AppCore* components, which are then executed in separate protection domains on the microkernel. Nizza bears some similarity with Proxos as it allows the *AppCore* components to communicate with and reuse the services of their untrusted counterparts.

Nizza and Proxos differ from each other in a number of ways. First, *AppCores* have to be modified to run on the interface of the microkernel while Proxos exports the same programming interface as the commodity OS, which facilitates the porting of applications. Next, Proxos does not require the manual splitting of the applications and existing applications can run on Proxos with minimal source code modifications. Even though it is a cumbersome task, the manual splitting of applications does provide developers with the advantage of a finer-grained control over what to include in an application's TCB. However, if not used judiciously, the *AppCores* could introduce heavy performance overheads. Applications are typically composed of logically related components. As demonstrated in our evaluation section, inter-VM communications are expensive. Nizza's manual splitting of the source code could potentially separate components that have a tight coupling

into different VMs, which would result in a lot of inter-VM interactions and a high performance cost. Proxos does not involve the splitting of applications and hence avoids such risks. Finally, Proxos and Nizza apply different approaches to distinguish between trusted and untrusted components. While Nizza splits the source code of applications into trusted and untrusted components, Proxos splits the interface between the application and the OS into trusted and untrusted components. Those two approaches are orthogonal and could be applied independently.

# Chapter 8

# Conclusion

Current commodity OSs export an interface that is too permissive to privileged applications, allowing compromised applications to gain control of the operating system kernel and attack other applications. Proxos allows applications to partition the interface between them and the commodity OS kernel into trusted and untrusted components by specifying system call routing rules. The end result is that Proxos allows application developers to protect applications from a compromised kernel without having to make major source code modifications.

By building a Proxos prototype and porting several representative applications, we have found that specifying trust at the system call interface is a powerful and simple way of isolating applications from the operating system. Proxos routing rule specifications are short and simple, and can be expressed in 10's of lines of code. Minor source code modifications are also required to support applications, mainly due to the semantics of the `fork` system call, and to remove any instances of dynamically loaded code that cannot be eliminated by static linking. In cases such as our web server, where expensive VM start-up and shutdown may become very frequent, further modifications are necessary to preserve performance. We expect that in most cases, a single graduate student who is familiar with an application can port it to Proxos in a day or two. With a modest

cost in engineering time and a reasonable impact on application performance, system call routing enables the developer to protect the secrecy and integrity of applications from a compromised operating system.

# Bibliography

[1] M. J. Accetta, R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. W. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of Summer Usenix*, July 1986. 1

[2] K. Asrigo, L. Litty, and D. Lie. Virtual machine-based monitoring of honeypots. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE 2006)*, June 2006. 2.4

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 164–177, October 2003. 1, 2.1, 2.2, 2.2.1, 4, 6

[4] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP 1997)*, pages 143–156, October 1997. 6

[5] CERT Coordination Center, 2006. `http://www.cert.org`. 1

[6] Christopher Clark, Keir Fraser, Steven Hand, Jakob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005)*, 2005. 2.5

[7] S. Devine, E. Bugnion, and M. Rosemblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. U.S. Patent 6397242, Oct. 1998. 2.2.1

[8] Dillo web browser, 2006. `http://www.dillo.org`. 5.1

[9] Petros Efstathopoulos, Maxwell Krohn, Steve Van De Bogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005)*, pages 17–30, October 2005. 7.2

[10] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP 1995)*, pages 251–266, December 1995. 7.1

[11] Norman Feske and Christian Helmuth. A Nitpicker's guide to a minimal-complexity secure GUI. In *ACSAC2005*, pages 85–94, December 2005. 4.1

[12] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the Workshop on Operating System and Architectural Support for the on demand IT InfraStructure*, October 2004. 2.2.2, 2.3.2, 2.6

[13] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 193–206, October 2003. 1, 2.4, 7.3

[14] S. Hand, A. Warfield, K. Fraser, E. Kottosovinos, and D. Magenheimer. Are virtual machine monitors microkernels done right? In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HOTOS 2005)*, June 2005. 2.6

[15] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of $\mu$-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP 1997)*, pages 66–77, October 1997. 2.6, 7.3

[16] Hermann Härtig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter. The Nizza secure-system architecture. In *Proceedings of the 1st International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2005)*, December 2005. 7.3

[17] M. Hohmuth and H. Härtig. Pragmatic nonblocking synchronization for real-time systems. In *Proceedings of the 2001 Annual Usenix Technical Conference*, 2001. 7.3

[18] Michael Hohmuth, Michael Peter, Herman Härtig, and Jonathan S. Shapiro. Reducing TCB size by using untrusted components - small kernels versus virtual-machine monitors. In *Proceedings of the 11th ACM SIGOPS European Workshop*, September 2004. 4.3

[19] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Analyzing integrity protection in the SELinux example policy. In *Proceedings of the 12th USENIX Security Symposium*, pages 59–74, August 2003. 1

[20] Xuxian Jiang and Dongyan Xu. Collapsar: A VM-based architecture for network attack detention center. In *Proceedings of the 13th USENIX Security Symposium*, pages 15–28, August 2004. 2.4

[21] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 Annual Usenix Technical Conference*, 2005. 2.5

[22] Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach. Analysis of an electronic voting system. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 27–42, May 2004. 1

[23] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 2001)*, December 2004. 2.2.2, 2.3.2

[24] J. Liedtke. On $\mu$-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP 1995)*, pages 237–250, December 1995. 2.6

[25] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the Linux operating system. In *FREENIX Track of the 2001 USENIX Annual Technical Conference (FREENIX'01)*, pages 29–42, June 2001. 1, 7.2

[26] Larry W. McVoy and Carl Staelin. LMbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Usenix Technical Conference*, pages 279–294, January 1996. 6.1

[27] Alex Moshchuk, Tanya Bragin, Steven D. Gribble, and Henry Levy. A crawler-based study of spyware in the web. In *Proceedings of the 13th Annual Symposium on Network and Distributed System Security (NDSS 2006)*, February 2006. 5.1

[28] Birgit Pfitzmann, James Riordan, Christian Stueble, Michael Waidner, and Arnd Weber. The PERSEUS system architecture. Technical Report RZ 3335, IBM Research Division, September 2001. 7.3

[29] S. A. Rajunas. The KeyKOS/KeySAFE system design. Technical Report SEC009-01, Key Logic, Inc., 1989. 7.2

[30] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium*, 2000. 2.2

[31] J. Saltzer and M. Schroeder. The protection of information in computer systems. *IEEE*, 63(9):1278–1308, September 1975. 7.2

[32] Stefan Saroiu, Steven D. Gribble, and Henry M. Levy. Measurement and analysis of spyware in a university environment. In *1st Symposium on Networked Systems Design and Implementation (NSDI 2004)*, pages 141–153, March 2004. 5.1

[33] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP 1999)*, pages 170–185, December 1999. 7.2

[34] Jonathan S. Shapiro, John Vanderburgh, Eric Northup, and David Chizmadia. Design of the EROS trusted window system. In *Proceedings of the 13th USENIX Security Symposium*, pages 165–178, August 2004. 4.1

[35] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing TCB complexity for security-sensitive applications: Three case studies. In *Proceedings of EuroSys 2006*, April 2006. 1, 7.3

[36] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006)*, November 2006. 1

[37] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005)*, pages 148–162, October 2005. 2.4, 4.2

[38] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, pages 181–194, December 2002. 2.2

[39] Webstone: The Benchmark for Web Servers, 2006. `http://www.mindcraft.com/benchmarks/webstone/`. 6.2

[40] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, pages 195–209, December 2002. 7.1