To be published in Proceedings of the 11th ACM SIGOPS European Workshop, Leuven, Belgium, 2004

Reducing TCB size by using untrusted components — small kernels versus virtual-machine monitors

Michael Hohmuth Michael Peter Hermann Härtig

Technische Universität Dresden
Department of Computer Science
Operating Systems Group

{hohmuth,peter,haertig}@os.inf.tu-dresden.de

Jonathan S. Shapiro

Johns Hopkins University
Department of Computer Science
Systems Research Laboratory

shap@cs.jhu.edu

Abstract

Secure systems are best built on top of a small trusted operating system: The smaller the operating system, the easier it can be assured or verified for correctness.

In this paper, we oppose the view that virtual-machine monitors (VMMs) are the smallest systems that provide secure isolation because they have been specifically designed to provide little more than this property. The problem with this assertion is that VMMs typically do not support interprocess communication, complicating the use of untrusted components inside a secure systems.

We propose extending traditional VMMs with features for secure message passing and memory sharing to enable the use of untrusted components in secure systems. We argue that moving system components out of the TCB into the untrusted part of the system and communicating with them using IPC reduces the overall size of the TCB.

We argue that many secure applications can make use of untrusted components through trusted wrappers without risking security properties such as confidentiality and integrity.

1 Introduction

What is the architecture of choice for building secure systems? Proponents of virtual-machine monitors (VMMs) are quick to point out that VMM technology leads to the smallest, simplest systems that are the easiest to assure. Certainly, a superficial analysis and historic evidence seem to support this thesis: VMMs are specifically designed to enforce isolation, the technology is mature, and of the four TCSEC/A1 systems, the cheapest to build and validate was the VAX/VMM hypervisor.

The basic assumption underlying this opinion is that the trusted computing base (TCB) of applications running inside a virtual machine cannot be made any smaller, as VMMs add to the bare hardware little more than isolation, hardware multiplexing, and trusted drivers, which constitute the basic mechanisms needed for confinement.

In this paper, we object to this view. We assert that by adding kernel features originally meant for extensible systems, such as message passing and memory sharing, the overall size of the TCB (which includes all components an application relies on) can actually be reduced for a large class of applications.

Another assumption we address in this paper is that all components on which an application has operational dependencies must be in this application's TCB. This presumption leads to the unnecessary inclusion of many (protocol and device) drivers into the TCB.

The basic idea for reducing TCB size is to extract system components from the TCB and consider them as untrusted without violating the security requirements of user applications. There are two basic techniques that facilitate this goal: trusted wrappers and inter-process communication (IPC), which comprises message passing and shared memory. Trusted wrappers encapsulate isolated untrusted system components and provide additional security properties. Memory sharing and IPC are microkernel-like mechanisms that enable efficient and controlled communication between isolated system components. Adding microkernel-like features (IPC) to VMMs may make the kernel more complex (first increasing TCB size), but enables using untrusted components outside the TCB, leading to an overall decrease in TCB size.

Additionally, by adding IPC system calls, emulation of communication devices (such as Ethernet emulation) for inter-task communication can be removed from the TCB.

Allowing untrusted system components enables the safe reuse of existing operating-system code. Reuse is generally desirable because it provides the functionality of a large body of known-to-work legacy code, and because it helps supporting backward compatibility with existing applications, data, or network protocols. Reuse is especially attractive for device and protocol drivers, as these components often make up the largest part of the operating system.[1] For example, device and protocol drivers make up more than 88 % of code lines of the Linux 2.6 kernel[2]. However, reused code is usu-

[1] Of course, drivers for devices that have unlimited DMA access always are a member of each subsystem's TCB. We return to device drivers in Section 3.3.

[2] This figure only counts lines that contribute to a x86 configuration. Irrelevant headers and architecture-dependent code are not considered.

ally untrusted—the point of reuse is not to look at (and not to worry about) the code. This motive is particularly present when reusing binary code for which source code is not available.

This work further develops a theme we introduced in our European SIGOPS 2002 paper "Security Architectures Revisited," in which we argued that secure systems should be based on small isolated components, and shortly outlined the reuse of untrusted legacy components through trusted wrappers (which we somewhat confusingly called *tunneling*) [7]. Since then, several authors proposed conventional virtual-machine technology without support for using untrusted components. The authors of Terra [4] even denounced microkernel technology as "exotic," implying that nothing can be learned from it. Therefore, in this paper we analyze the misconceptions that lead to this view and provide a blueprint for reducing TCB size.

This paper is organized as follows. In Section 2, we revisit the term "trusted computing base" to point out common problems in its use and to define it precisely. In Section 3, we explain how trusted wrappers work and in which scenarios they can help reducing TCB size. Section 4 compares VMMs and microkernels and identifies kernel services needed for the efficient support of untrusted components. We discuss related work in Section 5 and conclude the paper in Section 6.

2 "Trusted computing base": a revisitation

The term "trusted computing base" (TCB) has become an imprecise, often misused term. For example, its users often assume that there is only one TCB in a system. This is wrong because the word "trust" refers to relationships among components, each of which relies on a different set of components for its correct function. We refer to the set of components on which a subsystem S depends as the *TCB of S*.

We assume that adversaries can compromise untrusted system components, but not trusted ones.

It is illuminating to define precisely the meaning of the word "trust" in TCB. It refers to the assertion that the TCB fulfills certain specifications such as security, functional, and timing requirements. The security requirements fall into three mostly orthogonal main categories: confidentiality, integrity, and availability. In this paper, without loss of generality we subsume all functional and nonfunctional (timing, etc.) requirements under the "availability" label.

There is some divergent terminology in the security community about the definitions of the three security categories. Therefore, it is important to define them precisely and to point out potential misunderstandings. We use the following definitions (derived from security measures introduced in [10]):

Confidentiality: Only authorized users (entity, principal, etc.) can access information (data, programs, etc.).

Integrity: Either information is current, correct, and complete, or it is possible to detect that these properties do not hold.

Availability: Data is available when and where an authorized user needs it.

Especially our definition of integrity is inconsistent with that of some prominent authors, including Gasser's [5]: These authors define integrity to imply that data cannot be modified and destroyed without authorization. We refer to this property, which implies both integrity and availability according to our definition, as *recoverability.*

We deviate from the alternative integrity definition for two reasons. First, it creates overlap between integrity and availability, rendering the two categories nonorthogonal. Second, it is useful to reason about our (weaker) definition of integrity and about availability in isolation from recoverability.

The tools we have at our disposal for ensuring integrity are very different from the tools we use to ensure availability: In general, we can secure integrity (and confidentiality) using cryptographic means, whereas we establish availability (and recoverability) using software assessment and verification and—especially when establishing trust through software assessment is impossible or impractical (e. g., when using untrusted networks), or if hardware failures are part of the threat model—by introducing redundancy and using trusted backup media.

3 Trusted wrappers: Reusing untrusted components

For many applications, data confidentiality and integrity are vastly more important than availability; Gasser [5] conveys this observation as "I don't care if it works, as long as it is secure." Here are some examples: Remote–file-system users are happy with not trusting networks and disks as long as their data is backed up regularly (or permanently in a redundant disk array) and integrity and confidentiality are not at risk. Users of laptops and personal digital assistants (PDAs) are more ready to take the risk of having their mobile device stolen (rendering all data on it unavailable) if data confidentiality and integrity are ensured. People use legacy applications inside VMware on top of Linux, hoping that VMware ensures the integrity of their Linux systems.

In essence, for many applications it is acceptable to use untrusted components and to provide confidentiality and integrity in higher layers of a system. The important insight here is that trust dependencies are *not always transitive:* For instance, a subsystem that requires confidentiality from a component C does not necessarily require confidentiality from all subcomponents used by C.

We refer to components that provide security objectives for untrusted components using cryptographic means as *trusted wrappers.*[3]

3.1 Reuse example

Trusted wrappers enable the reuse of existing untrusted components in secure systems. To illustrate secure reuse, let us walk through a network-stack example. (For the moment, please ignore the dangers of not trusting drivers for devices with DMA capabilities—we will return to devices in Section 3.3.)

In this example, we consider the level of security to expect when accessing the Internet under the condition that the network stack is either trusted or untrusted.

Trusted network stack

Confidentiality: Yes—through cryptographic means

Integrity: Yes—through cryptographic means

Availability: No—because the Internet is untrusted

Even though the network stack is trusted, availability cannot be guaranteed to Internet users because network outages are common and because all network traffic will eventually cross a trust boundary, after which it will travel though the untrusted network stacks of routers that constitute the Internet. Additionally, Internet connections are susceptible to denial-of-service attacks.

When an application needs communication availability, it is forced to have available a trusted backup medium—for example, a private leased line or a direct USB connection to the communication partner—or a set of backup Internet connections.

Untrusted network stack Now let us extract the network stack from an application's trusted computing base and use a trusted wrapper component to provide confidentiality and integrity. Not having to trust the network stack allows us to reuse a robust, known-to-work implementation such as FreeBSD's or Linux's TCP/IP stack.

Confidentiality: Yes—encrypt data before handing it to the network stack

Integrity: Yes—sign data before handing it to the network stack

Availability: No—because the Internet is untrusted and because an adversary could compromise the network stack

In comparison to using a trusted network stack, the overall picture does not change. We still provide confidentiality and integrity, and availability is still missing at this layer. Again, applications must take extra precautions when availability is required.

This scenario involves one risk that is not present in the previous trusted-stack scenario: When an adversary compromises the untrusted network stack, she can potentially prevent any further communication with the application through the network (even if the network is completely trusted, which it is not in our scenario). Again, applications must shield against this attack using the standard means for providing availability: It needs to communicate using backup media, or using a redundant copy of the network stack, which in the simplest case might just be a rebooted instance of the networking software.

Restarting the network stack is relatively straightforward because it needs almost no persistent state. The situation is somewhat different for untrusted storage components. Here, an adversary could effectively and completely deny recoverability. The standard countermeasure for storage-availability problems is to back up all data to a trusted medium in regular intervals.

3.2 Perimeter versus sandbox wrappers

Using untrusted components is useful both within a secure system and on its perimeter, that is, at the interface to peripheral devices. These two modes impose slightly different requirements on the TCB, which we evaluate in this section.

In the first mode, *perimeter wrapping*, untrusted components never see unprotected (i. e., unencrypted) data. The untrusted software resides on the perimeter of the system, and trusted components use it to communicate with the outer world. This mode of operation is comparatively uncritical in that no special encapsulation of the untrusted component is required. The system must only provide for the unobservability of the actions and data of its trusted parts.

Sandboxing, on the other hand, works by using untrusted components internally to work on unprotected data, for example, for converting between data formats or for running legacy applications on classified data. This technique requires complete isolation for untrusted components, including measures to prevent open or covert communication to the outside world.

Another consequence of sandboxing is that there is no easy way for ensuring availability, which makes this method impractical when this property is needed.

3.3 Reusing legacy device drivers

As we explained in the introduction, the reuse of existing device and protocol drivers is especially desirable because of their sheer number and size. However, device drivers are special in that they access physical resources, which leads to the inclusion of most drivers into the TCB of all subsystems.

Drivers for devices that are capable of direct memory access (DMA) present the first fundamental problem: On systems without DMA protection, the driver can program

[3] In previous work, we have referred to the use of trusted wrappers as *tunneling*, in analogy to establishing secure channels over IP using IPSec tunneling.

the device to read or write to any physical memory frame, whether or not it is mapped into the driver's address space.

This problem is an instance of a more general problem: When a driver can access or allocate shared physical resources, its actions can be observed by all other drivers with access to the same resources, which allows for communication among all these drivers. Therefore, all drivers who have access to the same physical resources are part of the TCB of all subsystems using any of the drivers.

To remove drivers from the TCB of subsystem S, the operating system must provide complete memory safety including DMA protection, and it has to give resource-access guarantees the drivers used by S to avoid observability.

The current x86 PC architecture does not provide DMA protection. However, DMA protection is becoming more widespread: Some 64-bit chipsets (such as those designed for AMD's Opteron and the Alpha) provide a memory management unit for mapping 32-bit PCI-bus addresses to 64-bit memory-bus addresses and can be used for DMA protection on a per-bus basis. PCI Express will include a similar facility. PCs enabled for Microsoft's NGSCB will include DMA protection for a portion of the physical memory as well.

3.4 µSINA: Trusted wrappers in action

In the µSINA project, we have implemented an IPSec-based virtual-private-network (VPN) gateway that uses trusted wrappers to cut down the TCB [8].

For IP routing, this application uses two untrusted instances of L^{4}Linux, a port of the Linux kernel to the L4 microkernel interface [6], running as user-mode programs on one machine. While the Linux instances are untrusted, µSINA uses a set of trusted device drivers (running in their own address spaces). We have to trust these drivers because they have full DMA access.

The µSINA VPN gateway is designed to ensure confidentiality for communication across the Internet. To achieve this goal, we use a trusted cryptography engine and a trusted IPSec policy module. The cryptography engine serves as a trusted wrapper for the Linux instance used for outbound IP routing: It provides confidentiality and integrity for all Internet communication. (We currently do not wrap the other Linux instance, used for local IP routing, as our requirements did not call for integrity or confidentiality on the "inner network" better than that provided by Linux.)

The µSINA TCB consists of less than 25,000 lines of code, including the microkernel and the drivers. In comparison, the core of the x86 version of the Linux 2.6.3 kernel, *not including any drivers,* comprises more than 500,000 lines of code.

4 VMMs versus microkernels

In this section, we propose extending VMMs with kernel services known from microkernels to allow efficient use and reuse of untrusted components.

In the previous section we demonstrated how large, existing software components can be reused as untrusted subsystems in secure systems. There is, however, an important prerequisite: To be able to use untrusted components, the system must offer secure isolation of trusted and untrusted components, which in turn requires secure IPC for efficient cooperation among trusted and untrusted parts.

Traditional VMM systems are specifically designed for secure separation, but do not provide services for explicit memory sharing, and provide IPC only via emulated communication devices, whereas microkernels are designed for efficient communication and sharing.

To efficiently support untrusted components, we propose adding secure and fast IPC (including memory sharing) to VMMs. This addition first increases the TCB of all applications, but extracting subsystems from the TCB and morphing them into untrusted components results in an overall decrease in TCB size.

When IPC is available as an operating-system service, the emulation of communication devices does not need to be part of the TCB. IPC also enables fine-grained protected components, which is impractical when the unit of protection is a hardware-like virtual machine.

Traditional VMMs and microkernels can be thought of as the extremes of a continuum. Recent additions to this continuum are paravirtualized systems, such as Xen [1] and Denali [11], and systems that "virtualize" a user-mode–only process model, such as Fluke [3], which reside in proximity to traditional VMMs and classic microkernels, respectively.

We propose to allocate another spot in the center of this space: VMMs with microkernel-like features, or VM-enabled microkernels.

Table 1 summarizes the properties of the systems in the continuum. All system architectures in the design space do support reuse of untrusted components in some form. However, only microkernel-like systems enable their efficient employment through fast IPC.

µSINA on a VM-enabled microkernel If the L4 microkernel we used in our VPN implementation (Section 3.4) would include VM support, we could have used two unmodified Linux kernels instead of L^{4}Linux servers. We would have developed our IPSec IP-interface driver, which defers trusted functions to the external trusted servers via IPC, in the context of original Linux instead of that of L^{4}Linux.

5 Related work

Terra [4] has been proposed as a virtual-machine–based security architecture for trusted systems. Terra enhances traditional VMM technology with features for attestation, trusted user communication, and protection from administrators. Its authors describe the virtual-machine interface as a "small, stable interface" and imply that it can be implemented with 12,000 lines of code. However, drivers are clearly not included in this figure, and Terra has to fully trust them. Terra

Feature	**VMM**	**Para-VMM**	**VM-µK**	**user-VM**	**µK**
Process model	supervisor & user	supervisor & user	supervisor & user	only user	only user
Protection granularity	coarse	coarse	fine	fine	fine
High-performance IPC			+	+	+
Explicit memory sharing			+	+	+
Legacy reuse possible	+	+	+	+	+
Binary legacy reuse—user	+	+	+	+	+
Binary legacy reuse—kernel	+		?		
All drivers part of TCB	–	–			
Device emulation	+	+	+		
IPC needs device emulation	–	–			

Legend: "+" and "–" indicate features that are present. "+" stands for features that enable using untrusted components, "–" stands for features that hinder their adoption. "?" represents an optional feature.

Table 1: Features of VMM and microkernel systems

does not enable the reuse of untrusted components and does not offer IPC; communication between virtual machines is possible only using an emulated network.

Xen [1] is a paravirtualizing VMM. It supports a Linux kernel with minor modifications. The Xen kernel includes device drivers, which are controlled by privileged, fully trusted virtual machine. It currently does not support IPC or the use of untrusted components. Xen supports communication across protection boundaries using a virtual network device.

Nooks [9] is an architecture for using potentially erroneous device drivers within the Linux kernel. The encapsulation of these drivers ensures the integrity of the Linux kernel. However, Linux cannot protect itself against availability failures. The kernel has no means to preempt a driver that does not return control.

In earlier work, we have designed and implemented microkernel-based security architectures called Nizza that make extensive use of untrusted components [7]. Nizza is a general-purpose security architecture that uses L^4Linux as its legacy component. Like Terra, it provides attestation and a trusted path to the user using a small trusted windowing environment [2]. µSINA (which we discussed in Section 3.4) is a specialized implementation of the Nizza security architecture.

6 Conclusions

In this paper, we have proposed extending traditional VMMs with features for secure message passing and memory sharing to enable the use of untrusted components in secure systems. We argued that moving system components out of the TCB into the untrusted part of the system and communicating with them using IPC reduces the overall size of the TCB.

We argued that many secure applications can make use of untrusted components through trusted wrappers without risking security properties such as confidentiality and integrity.

References

[1] 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 nineteenth ACM symposium on Operating systems principles*, pages 164–177. ACM Press, 2003.

[2] Norman Feske and Hermann Härtig. DOpE — a window server for real-time and embedded systems. Technical Report TUD-FI03-10-September-2003, TU Dresden, 2003.

[3] Bryan Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullmann. Interface and execution models in the Fluke kernel. In *Proceedings of the third symposium on Operating systems design and implementation*, pages 101–115. USENIX Association, 1999.

[4] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 193–206. ACM Press, 2003.

[5] Morrie Gasser. *Building a secure computer system*. Van Nostrand Reinhold Co., 1988.

[6] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of µ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77, Saint-Malo, France, October 1997.

[7] Hermann Härtig. Security architectures revisited. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[8] C. Helmuth, A. Westfeld, and M. Sobirey. µSINA - Eine mikrokernbasierte Systemarchitektur für sichere Systemkomponenten. In *Deutscher IT-Sicherheitskongress des BSI*, volume 8 of *IT-Sicherheit im verteilten Chaos*, pages 439–453. Secumedia-Verlag Ingelsheim, May 2003.

[9] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 207–222. ACM Press, 2003.

[10] Victor L. Voydock and Stephen T. Kent. Security mechanisms in high-level network protocols. *ACM Comput. Surv.*, 15(2):135–171, 1983.

[11] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the fifth symposium on Operating systems design and implementation*, pages 195–209. USENIX Association, 2002.