Appears in the proceedings of CollaborateCom 2005, San Jose, CA, USA

The Nizza Secure-System Architecture

Hermann Härtig Michael Hohmuth Norman Feske Christian Helmuth
Adam Lackorzynski Frank Mehnert Michael Peter

Technische Universität Dresden
Institute for System Architecture
D-01062 Dresden, Germany
nizza-secarch@os.inf.tu-dresden.de

Abstract

The trusted computing bases (TCBs) of applications running on today's commodity operating systems have become extremely large. This paper presents an architecture that allows to build applications with a much smaller TCB. It is based on a kernelized architecture and on the reuse of legacy software using trusted wrappers. We discuss the design principles, the architecture and some components, and a number of usage examples.

1 Introduction

Desktop and hand-held computers are used for many functions, often in parallel, some of which are security sensitive. Online banking, virtual private networks, file-system-level encryption, or digital rights management are typical examples.

This type of computer use imposes two, often conflicting requirements: The mixed-use scenario usually necessitates the use of a full-featured, standard, general-purpose operating system (OS). On the other hand, security-sensitive applications must rely on, or *trust*, their operating environment to uphold the application's security guarantees.

Standard OSes have become so large that a complete security audit or a formal verification of security properties is absolutely illusory. This fact is illustrated by a steady stream of security-leak disclosures for all major operating systems. It seems that, for the time being, we just have to live with the bugs of standard OSes.

Therefore, while retaining standard OSes for their large set of features and applications, these OSes should not be part of a system's trusted computing base (TCB). In other words, security-sensitive applications should not have to rely on a standard OS (including the kernel) to assure their security properties.

To address the conflicting requirements of complete functionality and the protection of security-sensitive data, researchers have devised system architectures that reduce the system's TCB by running kernels in untrusted mode in a secure compartment on top of a small security kernel; security-sensitive services run alongside the OS in isolated compartments of their own. This architecture is widely referred to as *kernelized standard OS* or *kernelized system*.

In this paper, we describe Nizza, a new kernelized-system architecture. In the design of Nizza, we set out to answer the question of how small the TCB can be made.

We have argued in previous work that the (hardware and software) technologies needed to build small secure-system platforms have become much more mature since earlier attempts [8]. In Nizza, we put a number of these modern technologies to use.

In Section 2, we derive three construction principles from our requirements: isolation, trusted wrappers, and legacy reuse. Section 3 presents a motivational example use case for the Nizza architecture.

We validated our design by implementing a proof-of-concept prototype system with a set of secure components that can be combined flexibly according to application requirements. In Section 4, we describe the general architecture and four Nizza components in detail. In Section 5, we present two Nizza example applications we have built and evaluated.

Section 6 gives a survey of the technologies that enable the construction of Nizza.

We discuss related work in Section 7 and conclude the paper in Section 8.

2 Requirements and design principles

Nizza's basic architecture is derived from its two main requirements: a trusted system-software stack and compatibility with legacy software. In this section, we present three design principles we have derived from these requirements: small TCB consisting of isolated components; trusted wrappers; and reuse of legacy OSes.

Isolating security-sensitive code We observe that, for most applications, security-sensitive parts account for only a small fraction of the overall application complexity. As it is generally acknowledged that susceptibility to errors and attacks increases with complexity, we conclude that the vulnerability of the security-sensitive part can be decreased significantly by isolating this part from the security-insensitive part.

From these observations we derive the principle that the TCB should be minimized: Only essential security-sensitive functions should be part of the TCB. In other words, the TCB comprises only components that cannot be omitted without compromising the functionality and security of the service.

It is illuminating to define precisely the meaning of the word "trust" in TCB. It refers to the assertion that the TCB meets certain security, functional, and timing requirements. The security requirements fall into four main categories: confidentiality, integrity, recoverability, and availability.

Confidentiality: Only authorized users (entities, principals, 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.[1]

Recoverability: Information that has been damaged can be recovered eventually.

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

Along with the diverse demands of applications, the size and composition of the TCB varies. Thus, the TCB is application specific: It comprises the components a given application has to trust. Each trust relationship refers to one or more of the aforementioned security categories.

Components should reside in separate protection domains to prevent faults from propagating across component boundaries.[2] This isolation has to be complemented by a mechanism for controlled cross-domain communication (*inter-process communication*, or *IPC*).

As we mentioned previously, the number of errors correlates with the code size. Therefore, we favor a large number of small components over a few large ones.

Next, we describe a method to add functionality to an application without enlarging its TCB.

Trusted wrappers For many applications, data confidentiality and integrity (according to our integrity definition) are vastly more important than availability; Gasser [7] conveys this observation as "I don't care if it works, as long as it is secure." Here are two 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.

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 example, a component providing an encrypted file system can make use of an untrusted file-system component for the actual storage and provide integrity and confidentiality[3] itself using cryptographic means.

We refer to components that achieve security objectives for users of untrusted components as *trusted wrappers.*

In Nizza, we use trusted wrappers in secure applications to reuse untrusted device drivers, network-protocol drivers, or even whole legacy-OS instances. Section 4.5 presents the design of a trusted file-system wrapper, and Section 5.2 provides an elaborate example, in which trusted wrappers helped cutting down the TCB of a virtual-private-network router by an order of magnitude.

Reuse of legacy OSes To provide the full functionality of a standard OS, Nizza provides containers to securely run untrusted legacy OS components or even complete legacy OSes with their applications.

Nizza facilitates cooperation among security-sensitive and untrusted components: Legacy applications can use split-out trusted components, and trusted software can reuse legacy components through trusted wrappers.

[1]There is some divergent terminology in the security community about the definitions of the four security categories. Our definition of integrity differs from that of some prominent authors, including Gasser's [7]: These authors define integrity to imply that data cannot be modified and destroyed without authorization.

[2]Another approach to suppress uncontrolled error propagation is the use of secure languages throughout the system. However, this approach would impose severe implementation restrictions.

[3]Confidentiality is preserved as long as no unauthorized communication channels exist.

3 Example use case

To illustrate our approach for systems design, let us present a practical problem and a solution based on the mechanisms of Nizza.

People use commodity applications such as Mozilla Thunderbird for daily email communication. To provide a proof of the integrity of an email, a growing number of users sign their emails with the signing key as a credential. The confidentiality of the signing key is crucial for the user.

However, on today's commodity operating systems, the confidentiality of the signing key depends on the OS kernel including device drivers, root processes, the graphical user interface, the email program, the cryptography program, and the other processes of the user. For the example of Mozilla Thunderbird running on Linux, the crucial secret of the user is exposed to system components consisting of millions of lines of code. Still, people will not give up the functionality and convenience of such an application for security reasons. Using an alternative and more secure OS and a custom application instead of commodity software is not an option.

In current systems, security-sensitive and security-insensitive code often reside in the same protection domain. The presented principles of Nizza enable the drastic reduction of TCB size by moving security-sensitive functions from the commodity software to distinct protection domains, thereby eliminating untrusted code from the TCB. In fact, only the cryptography software, but no other system component, needs the signing key for proper operation. The isolation mechanism described in Section 2 facilitates the execution of the cryptography software inside a dedicated protection domain with exclusive access to the signing key of the user. In this example, we refer to this protection domain as *crypto domain*. Besides the crypto domain, we deploy Nizza's legacy container to execute a legacy OS and off-the-shelf email software. Both programs are executed on the same machine at the same time. When the user wants to sign an email, the email software has no access to the signing key. Instead, it hands over the email to the crypto domain using Nizza's secure communication mechanisms. In turn, the software in the crypto domain presents the textual information to the user and asks the user for approval to sign the email. When approved, the cryptography software signs the email and transfers the result back to the commodity email software for further processing. Therefore, the extremely complex commodity software can still be used without putting the user's credentials at risk.

Although this example provides an overview of how to use Nizza's mechanisms to improve security, it poses a number of new challenges: Both the commodity software as well as the software of the crypto domain must communicate with the user via a GUI. How can both parties securely interact with the user at the same time? How can we store the signing key of the user inside the crypto domain in a secure and persistent way? What engineering cost is required to make the commodity email program communicate with the crypto domain instead of performing cryptography by itself? How complex is the TCB of the crypto domain in this scenario? We will answer these questions in the following sections.

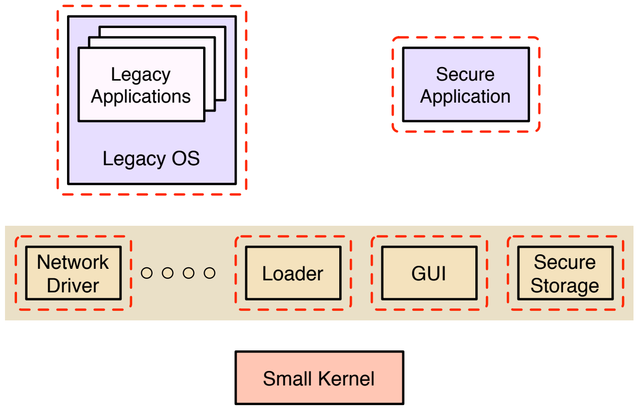

Figure 1. The Nizza architecture

4 The Nizza architecture

In this section, we describe the design of the Nizza secure-system architecture and the implementation of our Nizza prototype.

In Subsection 4.1, we present the general Nizza design. In Subsections 4.2 to 4.5, we elaborate on four components in detail: the kernel, the standard OS, the trusted GUI server, and a trusted file-system wrapper.

4.1 Overview

Figure 1 shows a sketch of the Nizza architecture. Nizza is composed of four major parts: a small kernel; a small secure-platform layer, consisting of trusted components (such as a loader and a trusted GUI component); security-sensitive applications; and an untrusted legacy standard OS with its applications.

The basic requirements for the small kernel are that it enforces component isolation in protection domains and provides fast communication between these domains (required for trusted wrappers and other secure-platform components). In Section 4.2 we report on our implementation's kernel.

Both the untrusted OS and the secure applications depend on the small secure platform, which provides minimal sufficient functionality for applications with high security requirements and ensures isolation.

The legacy operating system can be either an unmodified OS running in a virtual machine or a modified OS kernel adapted (*paravirtualized*) to run in a separate address space on top of the small kernel. We describe our legacy OS, L^{4}Linux, in Section 4.3.

In the following, we discuss three components of the secure-platform layer: the loader, the trusted GUI, and the secure-storage component.

The *loader* is the component responsible for loading and installing trusted components. It contains the machinery needed to load and establish applications besides and independent of the legacy OS. It has to make access-control decisions based on the system's policy. Additionally, it needs to support authenticated booting, which is the foundation for authentication chains. These unforgeable proofs of identity are made available to other components such as the trusted GUI.

A *trusted GUI* must establish the trusted path between the user and applications. It presents and uniquely labels (using authentication information) the screen output of multiple applications and protects user input from eavesdropping. We describe our trusted-GUI component in Section 4.4.

The *secure-storage* component, if it is designed for just confidentiality and integrity (but not availability), can rely on trusted wrappers. The actual storage of large data can be left to the legacy OS's file system. Cryptography can be used to protect data against information leaks and unnoticed modifications. It cannot protect against destruction of data, that is, against denial-of-service attacks. This may be tolerable if almost all data is stored on a server anyway, reducing the secure storage component to a mere cache for the largest part of data. However, direct (nonwrapped) secure storage is needed for the keys and for data added since the last backup. We discuss a possible file-service design in more detail in Section 4.5.

4.2 Fiasco microkernel

In our experimental Nizza implementation, we use the Fiasco microkernel [10]. It meets the requirements of a small kernel: Fiasco has been implemented in less than 15,000 lines of code. It is available on the x86 and ARM platforms, and for debugging purposes a version running on top of Linux is available as well.

Fiasco is an implementation of the L4 microkernel interface, a minimal kernel interface providing just three abstractions: address spaces, threads, and IPC [15]. On top of these mechanisms, operating systems can be constructed flexibly as a set of user-level programs. For example, the kernel interface allows memory management (including memory sharing and user-level paging), device drivers, and schedulers to be implemented on user level.

Nonetheless, Fiasco's L4 interface has a number of restrictions that must be overcome for a complete Nizza implementation: It currently lacks kernel-resource control and IPC control. Thus, untrusted components cannot be contained completely (see Section 6). Because of these shortcomings, we have started working on an improved version of L4 (nicknamed L4.sec) in which access to all kernel resources is controlled using kernel-protected capabilities.

4.3 L^{4}Linux

The key component for backward compatibility with existing commodity software is a container for executing legacy software. Depending on the software to reuse, the legacy software container of Nizza on Fiasco could be a language environment such as a Java virtual machine, a platform emulator such as Qemu, or a paravirtualized OS kernel.

In our experiments, we use L^{4}Linux—a paravirtualized Linux kernel—as the container for executing legacy software. L^{4}Linux is a modified Linux kernel that uses the L4 microkernel interface instead of accessing the hardware directly. Therefore, L^{4}Linux is executed without kernel privileges and cannot corrupt other system components that exist on the same machine. In [9], we showed this approach induces acceptable performance degradation. Our approach of modifying the Linux kernel's source code comes at the engineering cost of maintaining the changes of 7,000 lines of code. On the other hand, this approach does not require special hardware support or VMM primitives in the microkernel while providing excellent performance. The L^{4}Linux kernel is binary compatible with an unmodified Linux kernel and enables us to reuse existing Linux distributions and application software without any changes.

Applications such as the email scenario of Section 3 require secure communication between legacy software components and separate domains. L^{4}Linux supports hybrid Linux processes that can utilize both the Linux as well as the L4 system call interfaces. Such hybrid processes use L4 IPC to serve as the interface between the untrusted legacy container and trusted domains outside of L^{4}Linux.

4.4 Trusted GUI

In Section 3 we raised the question of how to enable secure user interaction with multiple applications. For the email scenario, the user needs to interact with the commodity email programs running on the legacy OS as well as with the crypto domain. For the commodity email program, we need the legacy GUI. Unfortunately, we cannot use the legacy GUI for the crypto domain as well because we must protect this domain from malicious code that potentially could gain control over the legacy GUI. Therefore, a separate trusted GUI component for establishing the trusted path between the user and the applications is indispensable.

EWM [22] and DOpE [5] follow the approach of implementing a custom secure host window system with a low complexity. The legacy GUI, including a complete commodity desktop environment, can be displayed in one single host window. This approach leads to fairly simple implementations of 5,000–12,000 lines of code. However, the features implemented by these window systems are not sufficient for the users who expect usage patterns of a wide variety of different commodity GUIs. For example, on Unix, people use and appreciate different X11 window managers with extremely different look and feel. Designing and implementing a GUI to fit all user's needs is not possible. With all the required features, this new GUI would become very complex. This is a conflict with our primary design goal of low complexity for this security-sensitive system component.

With our trusted GUI component Nitpicker [18], we follow the principle of exposing mechanisms but not imposing policy. Nitpicker only provides mechanisms for enforcing security and functionality that are crucial to implement a GUI in the client. There is no policy of window decorations, menus, and other GUI paradigms.

Nitpicker enables multiple client domains to share one screen while maintaining isolation between domains. It provides a virtual frame buffer to each client domain. Nitpicker segments the physical frame buffer into a set of regions, where each region is assigned to a corresponding virtual frame buffer. A client can configure regions that refer to its virtual frame buffer, but the composition of all virtual frame buffers on the physical frame buffer is only known to Nitpicker.

Each region on the visible screen is assigned to exactly one client. Nitpicker labels each region with the name of its client—obtained from the trusted loader—and thus provides information to uniquely identify each client on screen. This allows the user to detect attacks by Trojan horses.

For the communication from the user to the client domains, Nitpicker routes each input event to exactly one domain. The routing decision is derived from the current layout of screen regions, the mouse position, and a user-defined focused client. No client domain receives input events that refer to other domains. Spyware is not possible by design.

The mechanisms provided by Nitpicker enable us to use multiple legacy GUIs at the same time and integrate them into one desktop environment. All features, the look-and-feel, and commodity applications are preserved. Security-sensitive applications can interact with the user at the same time and only need to rely on Nitpicker. Our current implementation of Nitpicker consists of merely 1,500 lines of code. This is an order of magnitude less than the low-complex window systems mentioned before. As a current limitation, Nitpicker does not support hardware-accelerated graphics. Secure hardware acceleration is subject of our future work.

4.5 File-system design

A file system is a good example to demonstrate two important topics of the Nizza architecture, namely the reuse of untrusted legacy software and the dependency of TCBs on applications and their security requirements. In this subsection, we will present a back-of-the-envelope, as yet unimplemented design of a file system that can be configured either to meet just confidentiality and integrity requirements or to provide recoverability or availability as well. It leverages ideas of earlier work on the utilization of untrusted storage for trusted file systems (e. g., as in [13, 16]) for the preservation of confidentiality and integrity.

A key component of the secure platform in all configurations is buffer management for a flat file system. Upon read or write operations on files, it first inspects whether the respective blocks of a file are in a buffer. If so, it performs the operation using only trusted components. All other storage, that is, storage for files that are too large to reside in memory under control of the secure platform or that need to be persistent, is implemented using untrusted components.

If such a file system has to meet only confidentiality and integrity requirements (but neither recoverability nor availability; here called *simple configuration*), large parts of the storage reside in an untrusted file system that also provides persistent storage. As proposed in [16], cryptographic methods can be used to protect confidentiality, and hash sums, to protect integrity.

This approach requires only a relatively small amount of persistent secure storage in the secure platform for root keys and hash codes. Access to these keys must be provided only to trusted software that is authenticated by a trustworthy boot process. If there would be only volatile protected storage on the trusted side, replay attacks by replacing storage content and encrypted keys would become possible.

The simple configuration meets neither recoverability nor availability requirements. A successful attack that fully penetrates the untrusted legacy operating systems holding the large and persistent backup storage can cause data loss and corruption.

To additionally support recoverability, components have to be added to the file system to use trusted central servers to store either recoverable versions of the file system's state or at least credentials to recover such state from untrusted storage servers. The TCB then includes these additional components and the location of the trusted central server. The file system has to be augmented by a transaction-like scheme to identify checkpoints that are to be made recov-

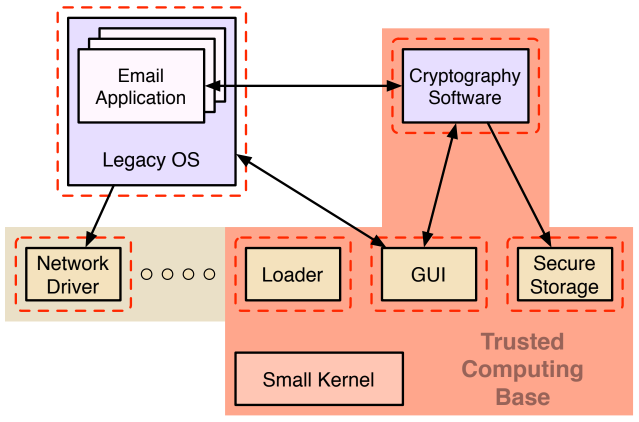

Figure 2. The system components of the email scenario. The confidentiality of the signing key depends only on the highlighted system components, but not on the highly complex legacy OS.

erable. The file system of the untrusted legacy system and the network protocols do not become part of the TCB of the file system. Thus, a successful penetrator of the legacy file system can prevent access to the file system.

To support availability in the presence of attacks, all components on which the file system operationally depends need to become part of the TCB.[4] In a system that needs to provide service in a local setting only, for example, for a patient monitor, the TCB increases by components managing the actual storage including disk drivers. It also needs to contain most if not all components that control power management.

5 Example applications

5.1 Email signatures

Our initial use case addressed the signing of emails (Fig. 2). This application moves security-sensitive functions from a commodity email client (Mozilla Thunderbird) to a separate PGP implementation.

Mozilla Thunderbird is a stand-alone email client that supports email signing via the third-party plug-in Enigmail. Enigmail uses the GnuPG program to read the user's private key and sign emails. It relies on Thunderbird to communicate with the user and maintain passphrases. Effectively, the proper function of Enigmail depends on the correctness of the email client and the multitudes of plug-ins that can be installed on the Thunderbird client to perform its security-relevant functions correctly. Thunderbird alone contains over 200,000 lines of code, rendering it virtually impossible to ensure correctness of the email client. As a side note, Thunderbird also shares some of the libraries with the Firefox browser (e. g., the HTML parser), thereby sharing their vulnerabilities, too.

[4]We ignore fault tolerance issues here.

With Thunderbird, the user signs an email by selecting the signing option before sending. When the user requests email transmission, the Enigmail plug-in is activated and retrieves the passphrase for the user's private signing key. Then Enigmail invokes the GnuPG program with passphrase and content of the email as parameters. GnuPG retrieves the key (e. g., from the hard disk), signs the email, and returns the result. The security-relevant data are the user's private key, the passphrase, and the content of the email.

To protect the confidentiality of the private key and passphrase and to allow the user a trustworthy review of the email contents, we move essential GnuPG functions, a content viewer and simple user dialog, to a separate protected domain—the crypto domain.

The crypto-domain component must be capable of performing three functions: First, display the content of the email in an unambiguous fashion for user review. Second, depending on the user's preference, sign the email with the user's private key. Finally, return the signed email to the email program. Note that plain-text content is available to the untrusted component even before signing. Therefore, the user must review the email in the trusted viewer. The key-management functionality from GnuPG must be included in the crypto domain because the signing algorithm needs the private key. The crypto-domain component displays the content of the email and waits for user input. Depending on the authorization by the user, the email is signed and returned to Thunderbird. Otherwise, it is discarded.

It is sufficient to store the PGP keys on standard storage devices. The confidentiality in case of data leakage is assured as the keys are protected by a passphrase. Nevertheless, saving the keys with the secure storage component increases the security and anchors the user's trust in hardware.

Section 3 raised the question of what engineering cost is required to securely reuse commodity email programs. Our solution only requires the user to configure Enigmail to use a forwarding proxy instead of the GnuPG program. The crypto domain and its TCB comprises about 105,000 lines of code, including Fiasco (15,000 lines of code), trusted L4 services (35,000 lines of code), and L4GnuPG (55,000 lines of code).

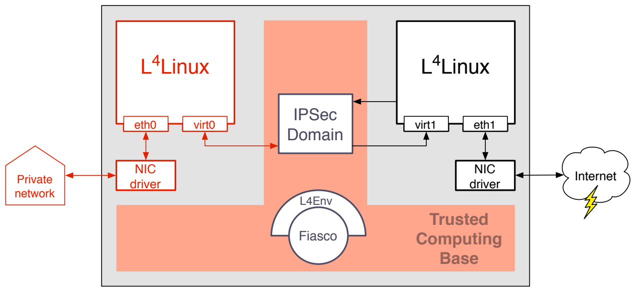

Figure 3. VPN gateway application.

5.2 VPN gateway

The majority of current VPN implementations are based on monolithic OSes. In monolithic OSes, the IPSec implementation is integrated in the kernel and closely interwoven with other components of the kernel, such as the network subsystem. Thus, bugs in the kernel code or a successful penetration of the complex monolithic kernel can compromise the security-relevant functions. The following example emphasizes the severity of the situation: In Linux 2.4, 70 percent of the kernel code are device drivers with an error probability 7 times higher than in other kernel modules [3]. This huge amount of code has unrestricted access to all data structures and functions of the kernel. A minimal configuration of the Linux 2.4 kernel comprises about 155,000 lines of code [23].

We observed that the security-relevant functions of a VPN implementation—data protection and policy enforcement—are only a small fraction of the monolithic kernel (less than 5 %). The Nizza architecture enables us to extract those IPSec-specific functions and execute them in a separate protection domain—the IPSec domain. This technique dramatically reduces the vulnerability of this sensitive functionality. The IPSec domain represents the actual connection point between the private network and the untrusted Internet.

Besides the IPSec domain, a VPN gateway requires network device drivers, IP packet processing including defragmentation, routing, and other basic networking functions for operation. All software components of the VPN gateway, excluding the IPSec domain, must be assigned to either the private or the Internet side.

Because both sides require general network functionality, we use two instances of L^{4}Linux running on one machine for providing these functions. Each of these L^{4}Linux instances is allowed to access one of the two physical network interface connectors (NICs) exclusively. Therefore, both instances are not able to communicate directly. The only way of passing data from either side to the other is the IPSec domain, which enforces the security policy and protects sensitive data. The scenario is depicted in Figure 3.

With the VPN software split into distinct components, we can revisit the security measures for each component individually to verify that sensitive information remains protected from unauthorized inspection and manipulation. The IPSec domain must be ultimately trusted regarding confidentiality and integrity of the processed data. Our implementation comprises merely 5,000 lines of code plus the used cryptography engine.

The L^{4}Linux instance on the Internet side never observes sensitive information because sensitive data is protected by the IPSec domain before leaving the VPN. Therefore, we can safely regard this L^{4}Linux instance as untrusted with respect to confidentiality and integrity of sensitive data. In contrast to the L^{4}Linux instance of the Internet side, the L^{4}Linux instance on the private side observes sensitive data. Nevertheless, network packets cannot leak to the Internet side because of the encapsulation. Thus, we do not need to trust this L^{4}Linux instance to meet our confidentiality claims. With respect to integrity of the sensitive data, the private L^{4}Linux needs to be trustworthy just as every component of the private network's infrastructure. The private L^{4}Linux instance cannot be attacked from the untrusted network because no unauthorized data from the untrusted network passes the IPSec domain.

In summary, the basic architecture of our VPN gateway reduces the TCB of the VPN gateway to the basic Nizza components plus the IPSec domain. Our VPN gateway comprises about 55,000 lines of code including the IPSec domain (5,000 lines of code). The highly complex functionality of a complete TCP/IP implementation is provided by untrusted legacy components without compromising our security objectives.

6 Enabling technologies

This section presents a survey of technologies that make the Nizza secure-system architecture possible.

Isolation The foundation of a minimal TCB is a small kernel, on which system components and applications run in their own protection domains.

It is conceivable that a Nizza system could be based on a virtual-machine monitor (VMM). However, contemporary VMMs depend on large components to provide memory management and hardware or device emulation for virtual machines. Currently, these components need to be fully trusted. It is an open research issue whether a small bare-bone VMM (or *hypervisor*) can be built in which security-sensitive services or applications do not have these management components in their TCB.

The isolation property also requires effective control of kernel objects and resources, such as memory and communication channels, for the following three reasons. First, it prevents denial-of-service attacks that lead to the consumption of all available resources. Second, it prevents unauthorized communication of untrusted components. This property is needed because otherwise containment of untrusted subsystems is impossible to prove. Third, it closes hidden and side channels that work by observing the resource consumption of another component.

One type of resource normally managed by kernels are user-visible names (or addresses) for kernel objects, such as memory addresses and thread IDs. The nonobservability requirement just mentioned can be addressed by localizing all names, that is, by never exporting global names (such as physical memory addresses) from the kernel.

Encapsulation relies on address spaces in microkernel-based implementations or virtual machines in hypervisor-based implementations. Both rely on a mapping of some form of virtual addresses to physical addresses. However, input–output (I/O) devices on current hardware (except some mainframes) that are capable of DMA bus-mastering use physical addresses to access memory. Thus, drivers using DMA or malicious firmware on devices can break encapsulation that is based on controlling the mapping of virtual to physical addresses. There are essentially two ways to enforce the mapping: DMA virtualization and I/O-TLBs.

DMA virtualization requires interception of critical operations of drivers and enforcement of correct behavior by emulation. This has some performance penalty and requires manual inspection of all drivers involved [17, 12].

I/O-TLBs are OS-controlled units between the memory bus and the devices. They already exist in rudimentary form in some systems such as AMD's Opteron, but provide only a single address space for I/O devices. Newer platforms are supposed to add more flexibility by providing dedicated TLBs on a per-device base.

Secure communication Strong isolation needs to be complemented with efficient and secure mechanisms for communication across protection domains (IPC) to enable controlled cooperation among system components irrespective of their trust relationship. Both low-latency and high-bandwidth communication is necessary—the former for remote-procedure-call-style invocations, the latter for streaming data (e. g., to peripheral-device drivers).

The literature provides many examples for both types of IPC (see for example [14, 20, 4]). From this body of work and our own experience we conclude that, for high-bandwidth IPC, the kernel should support sharing memory among multiple protection domains.

Support for standard OSes The only viable and secure way to provide complete compatibility with existing applications is to run the original OS, but in a sandbox that removes the OS from the TCB of security-sensitive applications and system components. There are two approaches for sandboxing an OS: full virtualization and paravirtualization. The first approach allows running an unmodified OS in a virtual-machine environment, whereas the second approach requires modifications to the original OS's source code because the paravirtualizing machine monitor does not export the exact original machine model. However, the simplified monitor interface is designed to reduce emulation cost.

The advantage of full virtualization is the possibility to run commercial OSes such as Windows. However, a VMM that does not inflate the system's minimal TCB with a large emulation framework has not been demonstrated yet. Therefore, Nizza currently employs paravirtualization.

Upcoming hardware automatically detects instructions that need to be emulated and traps into a hypervisor, eliminating the cost and restrictions of software-based workarounds. Virtualization support as offered in [11, 1] may require a reassessment of the situation. In principle, it seems possible to provide Nizza implementations that support unmodified legacy operating systems and still build applications based on very small TCB.

Authenticated booting and trusted platform modules Trusted platform modules (TPM) following recommendations of the Trusted Computing Group (TCG) provide two foundations needed for an implementation of Nizza: remote attestation and sealed memory (protected storage) [24], both based on authenticated booting. A TPM at startup time computes a hash value of the booted operating system and stores it reliably. The operating system may add other hash codes of loaded software components thus building a chain of authentication. This authentication chain can be retrieved using a challenge-response protocol and used for attestation. Additionally, the operating system can store keys inside the TPM that are bound to the booted configuration. This effectively enables the implementation of protected storage were bulk data is held on standard devices and secured (encrypted) with the keys under TPM control.

The TPM technology is essential for the Nizza architecture. It allows remote attestation. For example, it allows a client to check whether certain software is indeed used on the server and vice versa.

The basic layers of Nizza consist of the services described in Section 4.1, namely the small kernel, the loader, and the trusted GUI. A user-interface spoofing attack by manipulating a window manager to conceal that a malicious program (instead of the assumed trusted component) is controlling the display becomes impossible because it would leave detectable traces in the attestation chain.

A key assumption for the use of such technology is that the layers of software that form the authentication chain are hard to penetrate by an attacker. For instance, if the Linux kernel, the X Window System, Mozilla, and a Java virtual machine form the basis of a system, one has to assume that in the order of 10 million lines of codes can be sufficiently hardened, otherwise it is useless to include these components as basic parts of the authentication chain. In contrast, the TCB components of Nizza that constitute the chain are in the order of hundred thousand lines of code with a much better chance of protecting it from penetration.

7 Related work

The works closest to Nizza are Perseus [19] and Microsoft's plans for NGSCB (previously known as Palladium) [2]. Both system designs use a legacy operating system on top of small kernels and run applications with higher security requirements on top of these small kernels (e. g., *Palladium-enabled applications*). While Palladium planned for a kernel with virtual machine abstractions, Perseus has the same roots and is even based on some identical components as our current implementation of Nizza. To our knowledge, neither project has pushed the principle ideas as far as our experiments with Nizza. The notion of reuse of legacy through trusted wrappers, application-specific trusted computing bases, and minimized graphical user interfaces has not been extensively explored.

Terra [6] 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 does not enable the reuse of untrusted components and does not offer IPC; communication between virtual machines is possible only using an emulated network.

Another closely related and very interesting system is EROS [21]. Like Nizza, it is based on a microkernel and has a small windowing system component. It uses an elaborate capability system to control interaction between components. EROS supports persistent storage at a very low level, which is considered a source of great difficulties by the authors of EROS. So far, EROS does not consider extensive reuse of legacy a main engineering principle.

8 Conclusion

Trusted wrappers and the reuse of encapsulated legacy operating systems promise that complex applications can be built with very small application-specific trusted computing bases. An attractive property of Nizza is its supports hardware architectures ranging from main frames all the way to small embedded systems like cell-phone platforms. We have successfully ported an early L4-based (incomplete) version of a Nizza system to a small cell-phone platform.

The enabling technologies, namely small-kernel technology and novel hardware platforms, have matured tremendously in the past years. Early experience indicates that these promises are likely to be upheld. An experience paper currently being prepared will substantiate this claim based on elaborate analysis of several complex applications.

Acknowledgements

Work on Nizza is generously supported by grants from the Deutsche Forschungsgemeinschaft, the Bundesministerium für Wirtschaft und Arbeit, Intel Corporation, and IBM. We profited greatly from and enjoyed a meeting on a beach of Nice where both Perseus and Nizza started.

References

[1] Advanced Micro Devices, Inc. *AMD Secure Virtual Machine Architecture Reference Manual*, May 2005.

[2] A. Carroll, M. Juarez, J. Polk, and T. Leininger. Microsoft "Palladium": A Business Overview, Aug. 2002. Available from URL: `http://www.microsoft.com/PressPass/features/2002/jul02/0724palladiumwp.asp`.

[3] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, pages 73–88, Banff, Alberta, Canada, 2001. ACM Press.

[4] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)*, pages 189–202, Asheville, NC, Dec. 1993.

[5] N. Feske and H. Härtig. DOpE — a Window Server for Real-Time and Embedded Systems. Technical Report TUD-FI03-10-September-2003, TU Dresden, 2003.

[6] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. 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.

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

[8] H. Härtig. Security Architectures Revisited. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France, Sept. 2002.

[9] 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, Oct. 1997.

[10] M. Hohmuth and H. Härtig. Pragmatic nonblocking synchronization for real-time systems. In *USENIX Annual Technical Conference*, Boston, MA, June 2001.

[11] Intel Corporation. *Intel Virtualization Technology Specification for the IA-32 Intel Architecture*, 2005. Apr.

[12] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, San Francisco, CA, Dec. 2004.

[13] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 121–136, San Francisco, CA, Dec. 2004.

[14] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)*, pages 175–188, Asheville, NC, Dec. 1993.

[15] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, Dec. 1995.

[16] U. Maheshwari, R. Vingralek, and B. Shapiro. How to Build a Trusted Database System on Untrusted Storage. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 135–150, San Diego, CA, Oct. 2000.

[17] F. Mehnert. *Kapselung von Standard-Betriebssytemen*. PhD thesis, TU Dresden, July 2005.

[18] Norman Feske and Christian Helmuth. A Nitpicker's guide to a minimal-complexity secure GUI. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, 2005.

[19] B. Pfitzmann, J. Riordan, C. Stüble, M. Waidner, and A. Weber. Die PERSEUS System-Architektur. In D. Fox, M. Köhntopp, and A. Pfitzmann, editors, *Verlässliche IT-Systeme (VIS)*. GI, Vieweg, Sept. 2001.

[20] J. Shapiro, D. Farber, and J. M. Smith. The measured performance of a fast local IPC. In *5th International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pages 89–94, Seattle, WA, Oct. 1996.

[21] J. S. Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, Apr. 1999.

[22] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia. Design of the EROS Trusted Window System. In *Proceedings of the 13th USENIX Security Symposium (2004)*, pages 165–178, 2004.

[23] Snapgear. Snapgear Embedded Linux. URL: `http://ftp.snapgear.org/pub/snapgear/src/snapgear-3.2.0.tar.gz`.

[24] Trusted Computing Group. TCG Specification Architecture Overview, Apr. 2004. URL: `https://www.trustedcomputinggroup.org/downloads/TCG_1_0_Architecture_Overview.pdf`.