# A Survey of the Paper: *Terra: A Virtual Machine-Based Platform for Trusted Computing*

Gerald Stieglbauer

July 7, 2004

**Abstract**

Terra is an architecture for trusted computing using the concepts of virtual machine monitors to accomplish the needs of a wide range of security requirements. This article tries to summarize the situation found by the authors of the paper [7], which led to the development of Terra. Based on these studies, some arguments are presented, whether Terra gives an important contribution to this field or not.

## 1 Introduction

Today's computing environments (usually called operating systems) support a huge variety of different applications. Depending on their purpose of use, the applications have different requirements regarding performance, security and usability issues. For instance, consider a bank application, an online game or a web browser. Each of these applications has a different focus regarding the mentioned demands: All three applications have to provide a minimum of security, whereas the bank application has to be more secure than the online game. On the other hand, the bank application is a relative simple program with marginal performance requirements. In contrast, the online game needs high-speed graphic drivers on the local machine. To support the needs of all these applications, commodity operating systems (e.g. Microsoft Windows) are relatively huge software bundles with complex interfaces. Software, which is bound to these interfaces are

totally addicted to the operating system. Because of this consolidation of application and operating system, the reliability of software on top of an operating system depends also on the quality of the operating system. In practice, the quality of software is usually reciprocally proportional to its size. As a consequence, it is impossible to write simple, high-assurance applications for commodity operating systems.

However, vulnerable operating systems are not the only reason for vulnerable applications. Applications, which run simultaneous on a commodity operating system are poorly isolated. This means that an arbitrary (and potentially malicious) application can badly affect any other running application. Therefore, if we consider a set of running application, the whole system's security level is reduced to that of its most vulnerable application.

Sometimes, applications have to communicate with each other. The problem is how two applications can establish trust. Current platforms provide only weak mechanism for application authentication and attestation. Thus, an application has to assume that its communication peer is malicious. In addition, there is no trusted path between an application and a (human) user: The application is unable to recognize, if it is communicating with the user or with a (potentially malicious) program. Vice versa, the user has no warranties that he or she is really interacting with the application he or she believed to.

As a summary of the presented facts, computing environments have to be designed in order to

- are modular enough to tailor them as far as possible to the requirements of the application that should be supported (e.g. QNX).
- provide a strong isolation between concurrently running applications.
- support application authentication.
- be able to establish a (bidirectional) trusted path between the application and the user.

One common solution that addresses these requisites is a so-called closed system. In contrast to an open system, whereas the user has full control over the system, the control of a closed system is entirely given to the developer. The user has only restricted access to install new application or modifying operating system

components. Popular examples for closed box systems are cell phones or game consoles.

Nevertheless, both closed box systems and open box systems have a number of advantages and disadvantages. Terra tries to combine the advantages of both systems by using an architecture that is based on a virtual machine monitor. The authors of Terra argue that this approach is the best way to accomplish to two principles: *Isolation and attestation of applications.*

This article is structured as follows: Section 2 describes common techniques how application can be isolated to improve security. Section 3 is about the principles of application authentication and establishing trusted paths. Section 4 then describes the architecture of Terra, which tries to combine the principles presented in section 2 and 3. Finally Section 5 closes with a discussion about the main contribution of the paper.

# 2 Isolation of Applications

This section gives an overview about common strategies to improve isolation of applications compared to commodity operating systems. Each strategy belongs more or less to one or two of the following *system levels*:

- *Operating system level:* Integrating security policies into existing operating systems
- *Kernel level:* Providing alternative kernel architectures (e.g. micro-kernel)
- *Hardware level:* Providing a virtual hardware abstraction layer (e.g. virtual machine monitor)

In the following, we try to position each strategy somewhere in the suggested system level hierarchy.

## 2.1 Mandatory Access Control

Most operating systems rely on discretionary access control (DAC) mechanisms. These mechanisms distinguish only between two major categories of users: Completely trusted administrators and completely non-trusted ordinary users. DAC

access decisions are only based on user identity and ownership. Programs run by a user inherit all of the permissions of the user and have therefore access to all user objects, which are eventually used by other programs. Only weak protection against malicious software is provided.

To overcome that lack of application isolation, so-called mandatory access control (MAC) mechanisms are used. An example of a successful integration of such a mechanism is SELinux [12]. MAC access decisions are based on labels that can contain a variety of security-relevant information. The goal is a clean separation of the security policy logic and the enforcement mechanism. Each subject and object has a security label. All accesses from a subject to an object must be authorized by the policy based on these labels.

The MAC mechanism is an example of providing improved isolation by introducing enhanced security policies into an operating system. Kernel subsystems are used in SELinux in order to implement the clear separation of the security policy logic (security server) from the enforcement mechanism (object managers). In SELinux, the security server as well as the object managers are implemented as kernel subsystems. Regarding the three system levels, this means that the isolation strategy of SELinux is somewhere between the operating system level and the kernel level.

## 2.2 Isolating Drivers

Isolation of driver is not the primary topic in this survey. Nevertheless, in many cases, drivers are responsible for system crashes due insufficient isolation. Similar to the isolation of application, drivers, which use operating system interfaces too, have to be isolated to prevent them from affecting wider parts of the operating system and running applications. In [16], a reliability subsystem called Nooks is presented, which isolates driver from the operating system, without moving the driver code out of the kernel (cf. microkernels). This is achieved by introducing a reliability layer that is inserted between the driver and the kernel. As a consequence, existing kernels can be adapted with minimal effort.

The reliability layer has several purposes. The most important are:

- Isolation: This mechanism is used to prevent driver from damaging the kernel. Every isolated driver executes within its own lightweight kernel

protection domain. This means that the driver runs still with the same processor privilege, but has limited memory access rights.

- Interposition: This mechanism is implemented by some wrapper, so that the driver can be called transparently from a commodity kernel's point of view. In addition, in most cases a driver, which is called through this wrapper, does not have to be changed.

- Recovery: If a driver fails, this mechanism is used to simply restart the driver in order to avoid a crash of the whole system.

Nooks is a classical example of implementing isolation at kernel level. However, this approach focuses more on the isolation of drivers than of applications. Nevertheless, isolation of drivers plays an important role in the area of trusted computing.

## 2.3 Using Alternative Kernel Architectures

The strategies, which are presented in the previous sections, enhance more or less the capabilities of commodity operating systems and kernel architectures to implement isolation. However, there are some different approaches using alternative kernel architectures. Roughly, three different architectures can be distinguished:

- Microkernel architecture

- Exokernel architecture

- Isolation kernel architecture

In this section, the core principles of each architecture are described followed by some implementation examples.

### 2.3.1 Microkernel Architecture

In section 2.2 some additional layers are inserted into an existing kernel architecture to provide restricted memory access to kernel subsystems. Nevertheless, these subsystems still run in kernel privileged mode. The principle of a microkernel is to provide only the basic functions of a kernel[11]. All other subsystems are

moved out of the kernel and run in a lower privileged mode. In that way, faulty subsystems can not disturb the kernel core or other outsourced subsystems. In combination with mandatory access control mechanism (e.g. separating access control subsystems, see section 2.1), these subsystems as well as application using interfaces of the subsystems are efficiently isolated. In EROS [15] for instance, a collection of system services are implemented by non-privileged applications. The services provide higher-level abstraction such as files, directories and memory objects. These objects are administered by MAC policies, which define secure access control.

### 2.3.2 Exokernel and Isolation Kernel Architecture

Exokernels try to access the problem that commodity monolithic operating systems significantly limit the performance and freedom of application. This is because of traditional operating systems hide information about machine resources behind high-level core abstraction (e.g. the high cost of general-purpose virtual memory primitives [1]). This fact hinders applications to implement domain-specific optimizations.

Exokernels, however, securely multiplexes machine resources while permitting application-specific customization of traditional operating system abstractions [6]. On top of an exokernel, several non-trusted library operating systems using the exokernel interface. The exokernel interface isolates the library operating systems. The interface is very low-level and can be implemented efficiently. To ensure isolation, the exokernel interface must be secure, since resource control is given to the application as far as possible. The responsibility of an exokernel belongs to three major tasks:

1. Tracking ownership of resources
2. performing access control
3. revoking access to resources

The exokernel architecture is very similar to virtual machine monitors. Nevertheless, there are some differences: A virtual machine monitor tries to copies the underlying hardware. Copying, however, leads to hiding information and ineffective use of hardware resources in some cases [10]. Exokernel architectures are

optimized regarding performance, while virtual machine monitors are optimized regarding compatibility.

A similar approach is introduced by isolation kernels. Isolation kernels can be positioned somewhere between exokernels and virtual machine monitors. Like exokernels, isolation kernels do not copy the underlying hardware because of performance reasons. Nevertheless, the virtualization level of the underlying hardware is different to exokernels. The isolation kernel architecture Denali [17], for instance, exposes virtual, private name spaces, whereas exokernels expose the physical names of disk, memory and network resources.

## 2.4 Using Virtual Machine Monitors

Virtual machine monitors have a long tradition. First proposed and used in 1960 [9], several commercial applications (e.g. VMWare) based on this idea have great success. A virtual machine monitor is thin layer between hardware and operating systems. This layer provides several virtual machines, which, in contrast to exokernels, virtualize the real hardware as a one-to-one copy. This means that the virtual machine provide a hardware interface, which is (nearly) identically to the real hardware. As a consequence, commodity operating systems can run unmodified on this virtual hardware. Applications running on different operating systems, which themselves run on different virtual machines, communicate via virtual devices and are therefore clearly isolated. Like exokernels, security is guaranteed through relatively simple abstractions, such as a virtual CPU and memory[5]. Simple abstractions can be implemented by simple applications. Disco [4] is a good example for implementing a slim and efficient virtual machine monitor. Although exokernels are optimized for performance, the overhead of virtualization of a virtual machine monitor can be made negligible too [2]. As a matter of fact, an application's performance even can be improved, if a slimmer or more tailored operating system is used for a specific application[7]. Due to the benefits of a virtual machine monitor regarding compatibility issues, the system designer has a wider range of supported operating systems. In practice, this means that a system can be designed more rapidly with virtual machine monitors than with exokernel without surrender of performance improvement due the use of tailoring.

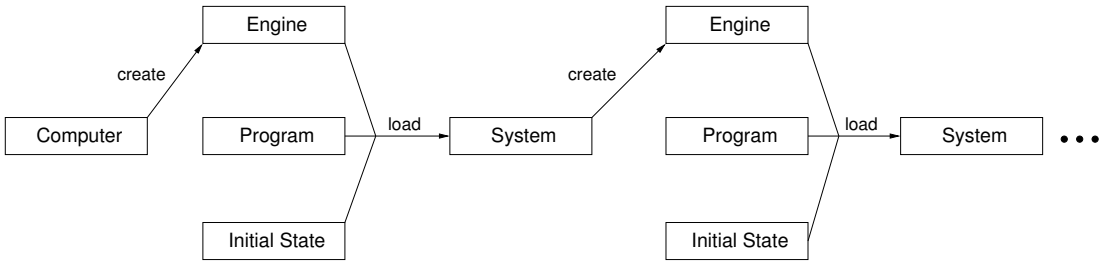


Figure 1: Computers, systems, programs and engines (adapted from [8])

# 3 Establishing Trust

Computer security is often reduced to two secure stand-alone operating systems, where secure connections are implemented between those systems. However, a secure connection has nothing to do with high-assurance applications, which are interacting with each other. A secure communication means that you are knowing either *who* creates the message that you want to read from a secure channel (authentication) or *who* can receive your message that you have written on a secure channel (confidentiality), or both of them (e.g. if a symmetric key channel is used). "Who" means in this context a system, represented by a unique ID (e.g. a network address). "Who" usually does not mean that you know with which application you are talking to. Therefore, sufficient techniques are needed that are able to establish trust between applications running on distributed or isolated systems.

## 3.1 System Authentication

Figure 1 sketches a generalized boot process. A computer is a special kind of a system and is made up by some hardware. A system is able to create an engine. In case of a computer, the engine is some boot code that runs on the computer. An engine usually loads a program with some initial state. The loaded and running program forms up an additional system. The system, which is loaded by the computer's boot code, is usually the operating system. At this point, the boot process does not stop: The operating system continues to load new systems called applications and so on. All loaded programs on a certain computer are called software stack.

To establish trust between two communicating computers, their software stacks have to be certified [8]. To implement this, on every computer each program is certified before it is loaded into the software stack. Certification means that the loaded program meets a given specification and can be assured that the system will behave as expected. Be aware that certifying a system does not have anything to do with software correctness! It only ensures, that a certain instance of software is loaded into the system (e.g. a certain Linux operating system, built on a certain date with a certain compiler).

During the system loading, a hash is created from the binary image of the program. This hash is signed with a private key used by the engine that loads the system, which results in generating a MAC (Message Authentication Code)[3]. The MACs of various images that may be loaded into a given system are collected in certificates. If the generated MAC complies with the engine's list of trusted applications it will be loaded, otherwise not. If it is loaded, the system receives the certificate too. When the system is asked to authenticate itself, it sends its certificate, which binds for instance an operating system to a certain computer.

Of course, certification is not only performed between a computer and an operating system. If an operating system is verified, a new private/public cryptographic key pair is created. The operating system uses the private key as a secret to sign for other systems (applications) that it loads. In that way, a certification chain is built, so that each system is uniquely bound to the software stack. Of course, if one of the systems is compromised after it starts nobody may find this out. This problem has to be considered as a separate issue and has to be handled outside of this technique.

## 3.2 Trusted Path

A trusted path is a mechanism by which a user may directly interact with trusted software, which can only be activated by either the user of the trusted software[13]. Without a trusted path, two situations may happen:

1. Malicious software impersonates the user to the trusted software. From the trusted software's point of view, there is no way to distinguish between an input coming from a user and an input coming from a malicious program.

2. Malicious software impersonates trusted software to the user. Sensitive

information given by the user to the malicious application can be used by the malicious software to break the system.

Commodity operating system hardly provides mechanism for trusted paths in the case of distributed applications. System authentication introduced in section 3.1 can be used to support the elimination of this lack of security mechanism: If an user uses a trusted application on the local machine to communicate with an certified application on a remote machine, a trusted path between the user and the remote application can be established (at least as long the remote application has not been replaced after building the certification chain).

# 4 The Architecture of Terra

This section explains how the Terra architecture tries to achieve isolation and attestation by using some of the previous approaches and techniques. Basically, Terra is a conceptional study about implementation guidelines for systems, which implement trusted computing. Although the authors have developed a prototype implementation according to these guidelines, this prototype only underlines the principles presented by the paper as a feasibility study.

To implement isolation of application, the authors have chosen the virtual machine monitor approach. They opted for this approach because of the characteristics of exokernels and virtual machine monitors regarding their clearer isolation abilities (mainly presented by this survey in section 2.3.2 and 2.4) compared to microkernels of mandatory access enhancements. Most of these abilities are shared by the exokernel and the virtual machine monitor approach. As an exception of this, the two approaches slightly differ regarding efficiency and compatibility issues. Here the authors voted clearly for the compatibility argument by using the virtual machine monitor approach. They argued that compatibility is more relevant in practice to improve the current situation of existing monolithic systems and the virtualization overhead is negligible even for virtual machine monitors.

The prototype implementation is based on the architecture, which is illustrated in figure 2: A VMWare GSX Server runs on the top of a Linux distribution. The VMWare Server and the Linux distribution have to be seen as the actual virtual machine monitor layer, which is indeed not very slim in the prototype

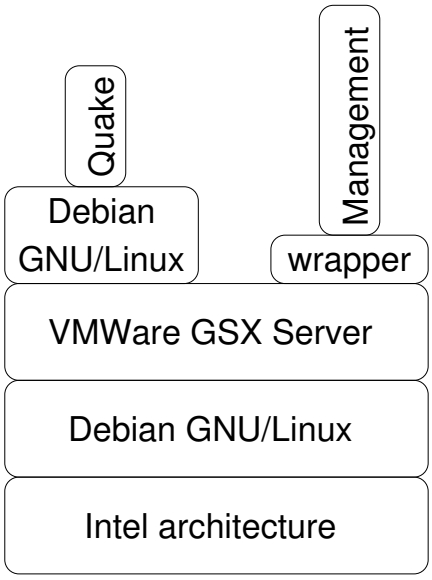


Figure 2: The architecture of Terra

implementation. On the first virtual machine, provided by the VMWare Server, another Linux system is running with only one application: An instance of the famous game Quake. The second virtual machine, a so-called Management VM is running. The Management VM is responsible for high-level configuration of the system, like creating new virtual machines, attaching them to virtual devices and so on. In case of the Management VM, the operating system is reduced to some wrapper calls, which simple forward commands from the Management VM to the VMWare Server.

The main contribution of the paper lies in the combination of isolation and attestation of application. To achieve that, the traditional virtual machine monitor architecture is enhanced by the following features:

- Authentication of application
- Trusted path
- Root secure

Authentication of application is based by the method described in section 3.1. Since the software stack differs from traditional systems, the software stack is built up as sketched in figure XXX: Instead of booting directly into an operating

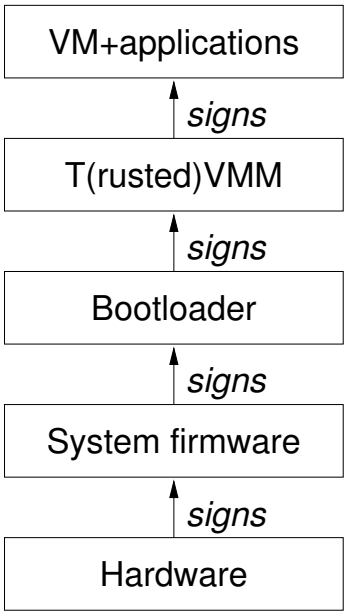


Figure 3: The certification chain

system, first the bootloader is signed by the hardware. The bootloader, in turn, signs the virtual machine monitor, which signs the operating system(s) and so on.

In the prototype system, of course, this certification chain is not fully implemented because of the decision of using existing architectures like the VMWare GSX server, which actually needs a host operating system to run. Therefore, the certification chain begins with the virtual machines provided by the VMWare server. The certification algorithm is implemented by some dynamically loaded libraries, which extend the VMWare API implementation.

The authors of Terra mentioned that trusted paths from the user to the application and vice versa have to be implemented. Nevertheless, this is only a postulation since they do not implement a secure user interface and refer only to existing approaches like the NetTop architecture [14]. Of course, the partly implemented attestation of applications of the prototype can play an important role in implementing a trusted path.

As mentioned in section 3.1, a successful certification of a software stack does not guarantee that some applications are modified or exchanged by malicious programs after the certification process. To access this problem, the Terra archi-

tecture offers two techniques. First, to perform integrity checks, communication between different layers of the software stack is partly realized using HMAC keys [3] (e.g. communication of applications via a virtual hardware device). Similar to the process of building a certification chain, this integrity checks are implemented by using dynamically loaded libraries. For instance, those libraries are used to overload the read/write operations used by the VMWare Server to access hardware devices.

The second technique tries to prevent the malicious exchange of running applications by guaranteeing root secure. Root secure means, that even the owner of the system has no access to the machine in order to break the basic privacy and isolation guarantees. This is closely related to the concept of a closed box. In contrast to closed system, in open system the platform owner has arbitrary access to the system. Terra supports both kinds of systems, in order to accomplish the security needs of the used applications: Open systems with highly non-trusted applications run in parallel to closed systems, whereas the open system has no chance to affect the closed system in any way.

In the prototype implementation, the game Quake is running at the top of a minimal Linux distribution to build up a closed box. The Linux distribution boots directly into Quake so that communication with the closed box is only possible by using Quake's network communication facilities. These are modified (again by the use dynamically loaded libraries) in order to perform attestation and key exchange with other clients of Quake on the same machine (but different virtual machine) or on a remote machine. To ensure that the owner of the system has no access to the closed box by removing some hardware (e.g. harddisk), encryption is used to store the image of the closed box on a certain hardware device. The modified read/write operations of the VMWare server are able to perform data encryption too.

# 5 Discussion and Conclusions

The paper *Terra: A Virtual Machine-Based Platform for Trusted Computing* describes a platform based on two principles: Authentication and isolation of applications. Various approaches exist in order to implement either authentication or isolation of applications. The main contribution of the paper is the concep-

tional combination of the two principles. According to the emergence theory, the whole is more than the sum of its parts. The paper tries to underline this by giving a broad context and suggestions how to implement the architecture. One strength of the presented approach is that it can be easily integrated into existing approaches. For instance, a standard Microsoft Windows environment has only weak mechanism for isolation and attestation. To run high-assurance applications, the Windows environment could be enhanced by using the principle of the Terra architecture as follows: A slightly modified commodity virtual machine monitor (e.g. a VMWare ESX Server that complies with Terra's postulations) providing a virtual machine is inserted as an additional layer between the hardware the Windows operating system. A second virtual machine is used to run a high assurance application (e.g. a bank application) on top of a rudimentary, high secure operating system. The high assurance application is running in a closed box, which is ensured by the virtual machine monitor. Because of multiplexing input events (e.g. mouse clicks) and output events (e.g. using special low-level video driver to display the GUI of the application in parallel with Windows applications), the user does not notice this isolation of applications. The secure application is able to connect itself to a bank server using authentication. Due the use of data encryption, the user or a malicious application that run under Windows has no chance to access the secure application to obtain any security information or to replace the secure application by a malicious program.

However, there are still some open questions or unaccomplished expectations. Although a broad context regarding how to implement a secure system is given, the presented prototype implementation is a little bit poorly. The authors of Terra claim to enhance existing solutions by providing root secure, attestation and a trusted path. The prototype implements no trusted path at all. Even attestation, which should be implemented by a chain of certification, is incomplete. Instead of building up the certification chain beginning with the hardware up to the application, the first link of the chain is the virtual machine provided by the VMWare GSX server. Since the GSX server needs a host operating system to run (e.g. Linux) no trusted base is provided. Therefore, the GSX server should be replaced at least by the ESX server, which is able to run directly on the hardware.

There are still many open questions how to build secure drivers and at which layer should they run. Although the paper addresses this issue too and refers

to some possible solutions, the problem itself is leaved as an open question. Regarding the presented demo architecture above, Terra offers no solution how to multiplex input and output events. For instance, video drivers are quite complex and low assurance applications in order to fulfill the requirements (e.g. performance issues) of various applications. Therefore, it is still an open question how to multiplex drivers for application with different needs.

Nevertheless, if Terra is seen as fingerpost to improve existing applications, the contribution of the paper is quite useful. The prototype implementation illustrates how to enhance an existing solution to improve isolation and introduce attestation of application to get towards a trusted computing base.

# References


[1] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 26, pages 96–107, New York, NY, 1991. ACM Press.

[2] 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.

[3] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Message authentication using hash functions - the HMAC construction. *CryptoBytes*, 2(1), 1996.

[4] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.

[5] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 133. IEEE Computer Society, 2001.

[6] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.

[7] 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 Symposium on Operating System Principles(SOSP 2003)*, October 2003.

[8] Morrie Gasser, Andi Goldstein, Charlie Kaufman, and Butler Lampson. The digital distributed system security architecture. In *Proceedings of the 12th National Computer Security Conference*, pages 305–319, 1989.

[9] R. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, (7):34–45, June 1974.

[10] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector Briceno, Russel Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Janotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Symposium on Operating Systems Principles*, pages 52–65, 1997.

[11] Jochen Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 237–250, december 1995.

[12] Peter Loscocco and Stephen Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, June 2001.

[13] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proceedings of the Nat. Inf. Sys. Sec. Conf.*, pages 303–314, October 1998.

[14] R. Meushaw and D. Simard. Nettop: Commercial technology in high assurance application. http://www.vmware.com/pdf/TechTrendNotes.pdf.

[15] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Symposium on Operating Systems Principles*, pages 170–185, 1999.

[16] 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.

[17] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: A scalable isolation kernel. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, St. Emilion, France, September 2002.