RZ 3335 (#93381) 04/09/01
Computer Science/Mathematics 15 pages

Research Report

The PERSEUS System Architecture

Birgit Pfitzmann[1], James Riordan[2], Christian Stüble[1], Michael Waidner[2], Arnd Weber[3]

[1] Universität des Saarlandes
Im Stadtwald 45
D-66123 Saarbrücken
Germany
{pfitzmann,stueble}@cs.uni-sb.de

[2] IBM Zurich Research Laboratory
Säumerstrasse 4
CH-8803 Rüschlikon
Switzerland
{rij,wmi}@zurich.ibm.com

[3] Karlsruhe Research Centre
Germany
Arnd.Weber@itas.fzk.de

The PERSEUS System Architecture

Birgit Pfitzmann
Saarland University, Germany
pfitzmann@cs.uni-sb.de

James Riordan
IBM Zurich Research Laboratory
rij@zurich.ibm.com

Christian Stüble
Saarland University, Germany
stueble@cs.uni-sb.de

Michael Waidner
IBM Zurich Research Laboratory
wmi@zurich.ibm.com

Arnd Weber
Karlsruhe Research Centre, Germany
Arnd.Weber@itas.fzk.de

Abstract

We present the system architecture and a prototype of Perseus, a secure operating system focusing on personal security management. Nevertheless Perseus allows users to use their favourite applications in a convenient, known way. It is built upon a trusted computing base that is small enough to be formally verified and evaluated according to the Common Criteria or ITSEC. The design includes the services necessary to support post-purchase installation of secure applications by the user. It is flexible enough to run on a wide range of hardware platforms, which allows PCs or PDAs to be used as general-purpose trusted devices. To support a common binary interface the Perseus system acts as a host that runs an existing operating system as one application (client OS). Moreover, by using the client OS judiciously to perform non-critical tasks, the size of the secure kernel can be significantly reduced compared to a stand-alone secure system.

1 Motivation

The advent of e-commerce, e-society, and e-citizenship has brought about the general need for end-user systems that can guarantee authenticity, integrity, privacy, anonymity, and reliability. While research on protocols, cryptography, languages, user interaction, etc. has provided solutions to a wide range of security related problems, all these solutions depend upon the proper functioning of the underlying system. To ensure such proper functioning, especially in a potentially adversarial setting, the need for secure underlying systems is clear.

Existing operating systems lack mechanisms to support security policies which are sufficiently easy to be maintained by any user, who likely is not a security professional. In addition to architectural insecurity and the inherent insecurity resulting from complexity, common operating systems require uniformly perfect and attentive system administration skills and will nevertheless not effectively protect individual users from executing malicious code.

Recent examples of application attacks are the VBS/Loveletter[1] and the Melissa virus[2] which take advantage of conceptual security holes of common applications to create significant damage.

Users execute foreign applets on their system and install software packages provided by unknown (and potentially malicious) parties without being able to decide whether their actions are security relevant or not. Applica-

[1]Estimated damage: \$4-12 billion in losses in as many as 20 countries [1].

[2]The ICSAs Tippett estimated that Melissa infected about 1.2 million computers and 53,000 servers at 7,800 North American companies that had at least 200 PCs, and it cost between \$249 million and \$561 million to fix (www.computer.org).

tions such as browsers act upon complex data sets, such as web pages, coming from untrusted sources. At the same time, users wish to use a single system for a wide range of purposes requiring high levels of security and yet belonging to different domains: financial, medical, professional, social, and personal. The open ended nature of our lives requires an open system.

As will be discussed in Section 2 security tools are an incomplete solution if the underlying components, especially the operating system, do not work correctly [6, 17]. We thus aim to implement a minimum security platform that fulfills all security requirements to protect users and their data from malicious code. We keep security-related components small enough to enable evaluation, based for example on the Common Criteria [18] or IT-SEC, and in a second step formally prove correctness. The Perseus architecture provides an environment to, e.g., create signatures securely without expensive hardware modifications. Additionally, it supports a client operating system (client OS) such as Linux or Windows to support mainstream applications.

The structure of this paper is as follows: The next section illustrates widespread security problems of common operating systems and builds a basis for Section 3 which presents security requirements to prevent those kind of attacks. Section 4 compares the Perseus approach and other ones that also try to build secure systems. Sections 5 and 6 present concept and design of security-related components, and in Section 7 a more detailed overview of the Perseus architecture is given. Finally, Section 8 gives a summary and points out further development steps.

2 Widespread Security Flaws

This section discusses the widespread, yet erroneous, belief that use of tools such as PGP, SSL, S/MIME or smartcards alone provides adequate security. We illustrate six scenarios that bypass these mechanisms by exploiting security flaws or architectural vulnerabilities of current operating systems. These scenarios provide a basis for a list of security requirements that have to be provided by secure operating systems, outlined in Section 3.

2.1 Mutual Protection of Applications

Common operating systems do not provide adequate mechanisms to protect different applications from one another. Thus malicious code (e.g., viruses, or Trojan Horses) is able to act with the full rights of the person executing them. Games have full access to financial information, financial programs have full access to medical information, and so forth. Malicious programs and data[3] are able to read, modify, and infect other programs and data. Virus scanners help diminish the effect of viruses but do not address the larger and more difficult problem of malicious software in general. Also, they typically only detect well-visible malicious code and are, of course, only reactive. But actually what is needed are proactive security mechanisms that are able to prevent installation of malicious code, because we must expect that future malicious code will be less "noisy", i.e. Trojan horses may attack single individuals or companies, without replicating themselves, and possibly even delete themselves, so that they will not easily be discovered by scanners.

2.2 Installation/Updates

It is very easy to attack a system by offering new applications, bug-fixes, or device drivers that come with a Trojan Horse, because there are no common mechanisms that guarantee correctness or limit application rights to the bare minimum needed.

Existing code signing mechanisms from Java or Active-X may allow users to determine where code came from but

- greatly restrict the class of software that can be executed, because they make it impossible to execute non-trustworthy code for convenience, inspection or just fun
- assume that non-malicious implies invulnerable, and

[3]Embedded macro and scripting languages increasingly blur the distinction between programs and data.

- require users to make decisions about social aspects of trust (these are generally more difficult than technical aspects).

2.3 Protected Paths

Another problem is the lack of mechanisms that allow users to realize which application they are communicating with (application authentication). This makes it very easy for malicious applications to deceive users, e.g., by displaying a faked dialog to enter a password [27]. Already Gasser proposed in [6] that a security kernel has to provide additional information to allow users to verify displayed contents to detect, faked dialogs for example.

A widespread approach to increase the security of existing computer systems uses smartcards or other trusted tokens. The idea behind this approach is to leave the operating system insecure and confine critical data (e.g. a private key) to a secure environment.

One problem of this mechanisms is that there is no trusted interface between the secure token and the user: the user has *no control* over what data is passed to the secure token [20]. It is thus possible for an attacker to instruct the operating system to modify documents between the secure token and the user: changing terms of a contract, generating bogus payments, signing false work orders, etc.

Devices that provide a trusted interface, e.g., a smartcard reader which comes with its own display, are not suited for general purposes and are unhandy. Smartcards with their own user interface are costly, and if one uses mobile phones as smartcard readers, it will be difficult to view normal business documents.

In addition to not being secure, a second, more limiting, problem exists: the mere storage and use of the private keys addresses only a tiny fraction of the functionality requiring security. Secure entry and editing, reading of private documents, and logging are all outside the range of what is possible with a security token no matter how tamper resistant it might be.

2.4 Insufficient Document Formats

Regarding digitally signed business documents, it is possible to maliciously alter the presentation, and hence meaning, of documents in many ways. General-purpose document formats are (appropriately) designed with utility rather than security in mind. This allows the creation of documents that display in one way in one environment and in a completely different way in another (so called smart documents). Even document formats not offering logical directive often display differently on different platforms, with different revisions of the software, with different system configurations, etc.

2.5 The Human Factor

The majority of users are not experts in computer administration or security (nor should they need to be). It is difficult — even for experts — to predict the *complete set of consequences* of any system change. Thus it is very difficult to decide whether a requested system modification is security relevant. An example is the seemingly harmless installation of a new font; this *is* security critical because it may change a contract for \$1,000,000 to one for £1,000,000.

2.6 Unsuitable Hardware

Even if the operating system is implemented and configured perfectly, it is possible to bypass the system security if direct hardware access is permitted to applications (as is often done for performance reasons). For example, an application with permission to access the harddisk controller is able to alter every memory address by using direct memory access (DMA) *without* going through the normal operating system access checks. This is clearly an enormous security hole.

3 Security Requirements

The preceding section has presented some example scenarios that show how easy it is to attack existing operating systems, as long as they do not provide adequate security concepts. We will now discuss a selection of fundamental security requirements that are essential to prevent those or similar attacks.

3.1 Secure Platform

It is a paradigm that security cannot be guaranteed without trusting at least one component. Thus, if a secure system shall execute potentially untrusted applications users have to trust at least some security-critical components. The smallest set of components users have to trust is generally called secure platform, secure kernel or trusted computing base (TCB). Obviously an operating system's secure platform has to be as small as possible to reduce the probability of faulty implementations and flawed assumptions.

3.2 Protected Domain

As long as untrustworthy or potential-incorrect software is used, the secure platform has to provide a mechanism to protect information of different subsystems from each other. It must, of course, protect itself against unauthorized modification. If such mechanisms are not supported by the hardware (e.g., memory protection mechanisms) interpretation of untrusted subsystems (e.g., by using a virtual machine) is the only protection mechanism. Because it seems impossible to enforce complex cohesions of trust using separated protected domains as provided by virtual machines and because virtual machines are very inefficient (especially untrustworthy applications such as games often require the most resources) we have to assume that the underlying hardware supports at least one protection mechanism. This mechanism protects both application code and data, to prevent attacks as outlined in Section 2.1.

3.3 Trusted Path

A trusted path is a mechanism allowing users to directly communicate with a specific subsystem and that cannot be altered or bypassed. If trusted paths are not offered by the TCB then security cannot be guaranteed.

Thus the TCB has to provide its own trusted communication path and use it to give users additional information which help to check the integrity of the displayed information. The simplest version of such a communication path has been suggested by Gasser [6]: a separated LED which indicates whether the user is communicating with the TCB. Clearly this solution is not sufficient for our purposes, because the trustworthiness of applications is context dependent and cannot be enforced by disjoint or hierarchical trust compartments.

Additionally, to guarantee integrity and confidentiality of inter-process communication (IPC) the TCB has to provide a protected communication channel between subsystems.

3.4 Access Control

To protect different applications from each other and to make it impossible for malicious applets or applications to obtain and then leak information, the system must prevent unauthorized manipulation of user data. It must also enforce a system-wide and configurable access control policy that enables fine-granulated and individual assignment of permissions to different compartments such as finance, health, game etc. Therefore, the system has to support two different access-control aspects:

1. Mandatory (system-wide) access control providing a fail-save structure which enforces that the system remains in a secure state[4]. The rules of this part of the access control should be hardcoded or only be maintainable by professionals.

2. Discretionary access control which is sufficiently fine granulated to allow an entire whole range of security policies to be defined. The system has to provide an easyly understandable configuration tool that allows non-experts intuitively to define their vague knowledge about trust using a high-level language.

To be able to enforce a wide range of security policies the access control mechanisms should be able to control every interaction between subsystems.

[4]What a secure state is has to be defined by the security policy.

3.5 Installation/Update Service

To prevent users from compromising security of their system by installing or updating applications, and to allow a given security policy to be enforced, a trusted service has to observe these operations. The service's task is to derive the application's permissions depending on a given security policy. Furthermore it has to choose the destination compartment and update the access control database. Because it is unlikely that these operations can be performed without user interaction, the service has to offer an easy to understand, high-level user interface.

3.6 Hardware Encapsulation

Section 2.6 mentions that subsystems which can access the hardware are able to put the system into an insecure state. Three approaches exist to prevent attacks of this kind:

1. Allow only the TCB to access the hardware.

2. Use only hardware that, e.g., does not support DMA.

3. Improve the hardware design to prevent security holes, e.g., by developing DMA devices that use virtual memory.

We do not expect appropriate hardware improvements in the near future, thus we focus on the first two options. If only the TCB is allowed to access the hardware all device drivers have to be part of the secure platform, which increases its size (a SCSI driver has about 50000 lines of code) and makes verification and evaluation harder. Also, there is no guarantee that our driver is more stable than the one provided by the client OS, except that our driver runs within its own protected domain, which prevents interferences with other subsystems.

To use only hardware devices that fulfill our security requirements seems to be another promising solution. Especially if it is possible to develop a TCB for a homogenous and well-known hardware environment such as some sort of PDA, it should be possible to keep the TCB very small.

3.7 Verification and Evaluation

State-of-the-art development of security-related software should include a system specification, implementation, as well as the proof that the implementation matches the specification, to increases its reliability and trustworthiness. Experiences of other projects [22, 25, 26] have proven that is it possible to verify mid-sized software using modern tools like PVS [5] or VSE [2]. An important objective of the Perseus project is to develop a completely verified security platform. To increase the assurance level we aim to evaluate our specification and design of security-related components using the Common Criteria [18] or ITSEC.

In the cryptographic community a security requirement is to provide an open design, and in our opinion open source and open documentation can further increase the trustworthiness of software. Therefore we decided to make the source code and design steps publicly available and to put the code under the GNU Lesser General Public License LGPL[5].

3.8 Trusted Devices

Security can only be guaranteed as long as the hardware is unmodified. Personal mobile devices such as PDAs or mobile phones are an attractive platform, as they are most of the time under control of their owner, which makes malicious modifications more difficult. Normal PCs, however, are also important platforms as important business documents are usually handled on such machines and laptop owners, for example, can exercise a certain control of their hardware.

Furthermore, implementation on a mobile device has the advantage that it provides a homogenous hardware environment (see Section 3.6). The functionality and speed of these devices should suffice, especially if cryptographic operations such as key generation and signing can be offloaded into smartcards.

Our security analysis uncovers security-related problems of existing hardware components, therefore we will be able to suggest improvements to prevent those problems when developing a highly trusted hardware basis (see

[5]http://www.gnu.org/copyleft/lesser.html

[12], where it is shown that it is possible to prove correctness of a microprocessor, but at the moment without considering security requirements). This will make it possible to use a completely verified system in highly security-critical applications.

3.9 User Friendliness

As explained in Section 2.5 it is very important to consider user behaviour and knowledge when designing secure systems. Another important aspect is that security functions should work transparently whenever possible. This prevents that users will find a way to bypass them because of simplicity. Nothing is more dangerous than a security mechanism that has been turned off by the user.

Analysis of user behavior and evaluation of current problems will enable us to develop a system that provides a failsafe structure and prevents users from inadvertently creating security holes.

4 Related Work

Because public awareness of the need for computer security within the scientific community is not new, different approaches are used to develop secure computer systems. This section gives a short introduction to some these efforts and compares them with our approach.

4.1 New Systems

There are a few operating systems that have been developed from scratch by considering security requirements during all development steps. Examples are the Birlix [11], Multics [4] and Hydra [28] operating system, more recent ones are the capability-based EROS [21] and SPIN [3].

This approach provides the most flexible way to obtain a secure operating system, but requires huge resources because not only the operating system but also all user applications have to be developed from scratch. Moreover, users have to learn new concepts, and developers have to acquire new interfaces and environments. Thus the main disadvantage of this approach is that compatibility with widespread operating systems cannot be provided. This may be a reason why a large number of secure operating systems live in obscurity.

4.2 Improvements

Another common approach increases the security of a common operating system by extending only some security functions. An incomplete list is RSBAC-Linux[6] [19] and the SecureLinux[7] project. The advantages of these approaches are obvious:

1. Compatibility with existing binary interfaces can be retained, allowing existing applications to be reused.

2. Relatively few modifications of the operating system are necessary, the largest part of the existing system remains unchanged.

Note, however, that these approaches will only increase the security of the system, they will never provide high-level security without a complete rewrite. Because the security of the extentions depends on the correctness of the entire kernel, including all drivers and modules (there is no protection mechanism between them), it is improbable that these systems will provide security on a high assurance level in the future.

4.3 Microkernel Approaches

Some projects try to overcome the disadvantages mentioned in Section 4.2 by developing a multiserver operating system that provides a binary-compatible interface to a common operating system. Examples are SawMill-Linux [7] based on the L4 μ-kernel interface and Flask [23], a Mach-based system.

As far as we know these approaches neither aim to keep security-related parts as small as possible, nor do they provide mechanisms such as trusted path or protected compartments.

[6]http://www.rsbac.de

[7]http://www.nsa.gov/selinux/index.html

5 General Concepts

The concepts of the Perseus project are as follows: Based on a μ-kernel that guarantees elementary security properties we build a minimal multi-server-based security platform. On top of this interface, a common operating system (client OS) provides binary compatibility for non-critical tasks. Security-related data and functions are extracted to applications running in parallel to the client OS, protected by mechanisms of the security platform. Figure 1 gives an overview of the three main system components:

The so-called red line separates potentially untrusted subsystems and the secure platform, which contains all security critical components such as access control and/or information flow mechanisms, low-level drivers, the user interface and smartcard reader support. All subsystems above the red line are under control of these mechanisms and unable to bypass security mechanisms or put the system into an insecure state. Therefore as long as insecure devices are used (see Section 3.6), only subsystems below the red line are allowed to access the hardware directly.

The secure platform is built upon the hardware platform (below the yellow line), which has to provide at least one mechanism (see Section 3.2) to protect different tasks from one another. Currently we assume that the hardware works correctly, but we hope to be able to support verified hardware in a later development stage.

Because the platform provides only low-level device driver support we keep it small enough to be verifiable and to run on mobile devices. It acts as a host system and executes one or more tamed client operating systems (e.g., Linux, Windows or EPOC) on top of it. Of course this approach does not increase the security of the client OS, but by extracting security-critical data from the client OS and providing secure applications that run directly on top of the secure platform, new security mechanisms are able to protect those data and enforce their own security policies.

The multi-server approach provides a flexibility that can be used to adapt the system to different hardware platforms and to use it for various purposes. For example, running it on a PDA will neither require services to access harddisks nor persistency in a different manner because PDAs come with static memory that keep the PDAs state even if the system is powered off. However, another graphic driver is required that takes into account the different screen size. Running a reduced version on a simple PDA without client OS produces a verified smartcard reader which provides a graphical display. In this case, the hardware does not even have to provide a memory-protection mechanism because only trusted software is used.

The flexibility provided by the microkernel also enables switching between different service implementations on demand, e.g., to replace a fast signing service by an evaluated one. For flexibility reasons we have to prevent dependencies between the different services. Thus system services are defined by interfaces and implementations communicate to each other only via those interfaces using a naming service that resolves requests [24]. To support updates of interface definitions and to be able to distinguish different services implementing the same interface, each service is precisely identified by a triplet consisting of an interface, a version and an individual name.

5.1 The Client OS

As initially mentioned, the Perseus system supports compatibility to a common ABI (Application Binary Interface) by running another operating system as client. This can, e.g., be Linux or a Windows OS, which allows users to use their favourite word processor in a convenient, known way. Only security-critical operations such as signing are performed by secure applications, hence inaccessible to Trojan horses or administrators. Regarding signing, the text would be displayed securely by a trusted document viewer before it is signed. For convenience, the signed text, and the original Word processor text, can then both be sent to the recipient.

For supporting a client OS, three alternatives exist. First, the existing OS can be ported to the interfaces provided by the secure plat-

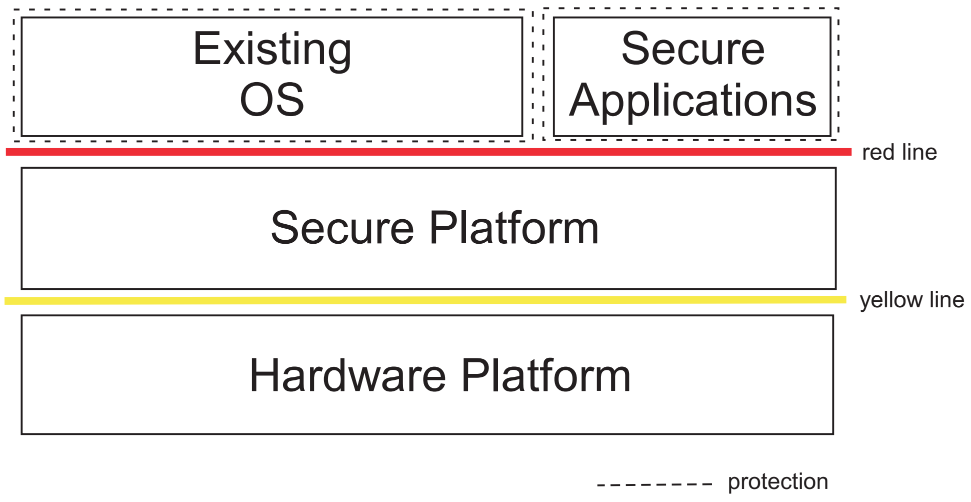


Figure 1: General design model of the Perseus architecture.

form, which has already be done with Linux, but is a problem if the source code of the OS is not available (e.g. Windows). Second, an existing virtual machine such as VMware[8] or its free clone Plex86 [9] running on Linux can be used to execute any machine-compatible operating system. As this will slow down performance, it will be more advisable to have a virtual machine directly running on the kernel, e.g., by adapting Plex86 to the interfaces of the secure platform.

third, to process critical data using unsecure applications, e.g., to write a text confidentially, the user can execute several instances of the client OS. For example, users can run one instance to perform only critical applications such as a word processor to enter critical data or online banking software, and another instance to for running untrustworthy code such as web browsers, downloaded contents or games.

In other words: with our approach, users will have a possiblility to protect legacy applications. They also have an opportunity to run arbitrary code. In the long run, they get an opportunity to have newly designed, highly secure applications running in a secure compartment. Of course, they should not re-open secure applications for running non-trustworthy scripts etc. Such code should be confined to the surf-and-play compartment.

[8]www.vmware.com
[9]www.plex86.org

6 The Secure Platform

This section gives a brief introduction into security-related concepts of the secure platform, outlined in Figure 2.

A security architecture is an abstract machine that describes overall security concepts of the TCB to meet security-related requirements and defines its behavior in an abstract manner.

The secure platform is an instance of the security architecture containing all security-relevant subsystems and enforces a given security policy. Because subsystems of the secure platform cannot be controlled by mechanisms other than those provided by the hardware (if they could they should not be part of the secure platform), users have to trust them completely. Therefore the most important requirement is to reduce the complexity of the security architecture and its subsystems to decrease the possibility of errors and to facilitate later evaluation. The secure platform itself is divided into different parts, which will be explained in the following subsections.

6.1 Hardware Abstraction

To keep the security platform as hardware independent as possible, the secure platform contains a layer providing an abstract view to the hardware. These are the μ-kernel, which runs

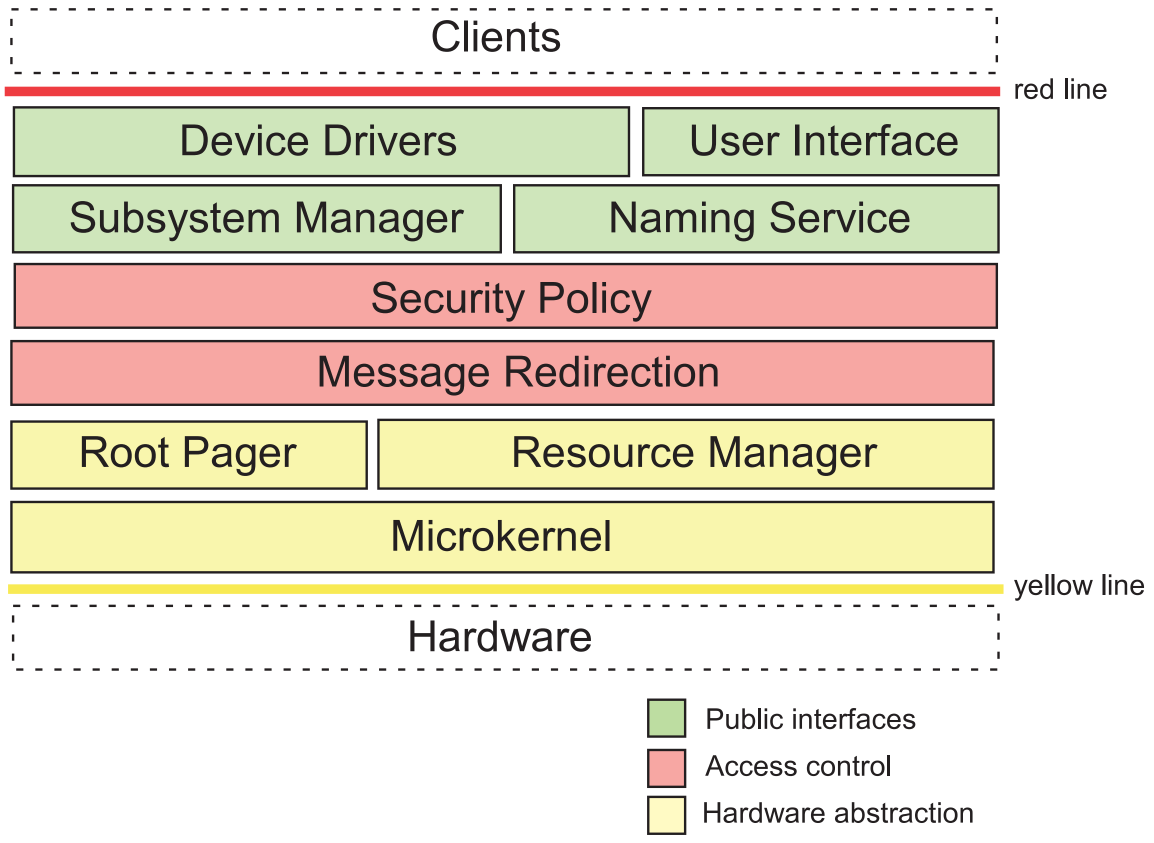


Figure 2: The different layers of the secure platform.

on top of the CPU and provides an IPC mechanism, an abstract view to pagetables called address spaces, and mechanisms to manage tasks and threads. A μ-kernel is used rather than a monolithic kernel because of its reduced functionality and complexity. The latter is very important for our purposes, because the operating system kernel is the most security-critical part of an operating system (it can access every data structure), and large, monolithic kernels are more difficult to maintain and to evaluate.

The μ-kernel also has to provide a message redirection mechanism (see Section 6.2) to implement a reference monitor enforcing a system-wide security policy. All other subsystems except the μ-kernel are executed in user-mode. Beside the μ-kernel several services provide an abstract view to hardware resources such as interrupts, I/O ports and DMA channels and one or more root memory pagers, which provide a persistent view to the subsystem's address space.

The use of persistent virtual-memory addresses has three important advantages: First, it prevents having to implement an entire file system or database within the secure platform, which would increase its complexity significantly. Second, it provides a unified view to non-persistent systems such as desktop PC's and persistent ones like PDA's. Third, this design guarantees that a task's data never leaves its address space (e.g. to store it into a file) and therefore the protected domain. Because the only way information can leave its protected domain is IPC, we can guarantee that a security policy can be enforced by controlling IPC.

6.2 Access Control

Because the granularity of the mandatory access control has a service-based granularity and because services are implemented by threads, the reference monitor has to provide a thread-based granularity. The policy defines a matrix M which decides whether a thread t_i has the permission to access thread t_j via IPC. The implementation will depend on the redirection mechanisms provided by the μ-kernel. A more general overview is given in [13], and Section 7.2 describes an implementation based on clans and chiefs [16].

6.3 The Subsystem Manager

The subsystem manager is a service taht derives the permissions of new applications based on a given policy. Therefore users have to invoke this service to install or update applications. To prevent users from defining access control rules to every application and to simplify the policy, database applications are separated into different classes, called compartments. The security policy therefore defines rules which are evaluated by the derivation service to determine which compartment an application is assigned to (e.g., game, health, and bank) and which permissions are granted. A mechanism used by the derivation service could be certificates of trusted third parties that guarantee correctness of the downloaded content [14].

6.4 Device Drivers

To reduce the number of covered channels and to prevent attacks as outlined in Section 2.6, the secure platform has to contain drivers for all devices that provide mechanisms that could be used to disturb the security of the system. These drivers also provide mechanisms to share hardware between different client applications, e.g., the client OS.

6.5 User Interface

The user interface is a collection of highly security-relevant device drivers, because it encapsulates accesses to user input and output devices. It has an event-based structure, but the implementation of core components depends on the underlying hardware, screen size, and available input devices provided by the device. This service also has to provide a protected path as defined in Section 2.3. Gasser originally suggested to use a "secure attention signal" for allowing the operator to tell between secure and other applications. On a mobile device, this could be a diode, showing a green light when a certified application is running. On a PC, a separate display could provide more information, such as the name of the manufacturer, or information about the quality of the signature under the code. Alternatively, a protected area of the screen can be used for this purposes (see for example Figures 5 and 6), which should be no problem on large-sized PC displays, but could be a problem on small-sized PDAs or mobile phones.

7 Implementation

This section gives an overview of the first Perseus prototype, outlined in Figure 3. The modules have been implemented using the C++ programming language.

7.1 Hardware Abstraction

We use the Fiasco [10, 15] μ-kernel, which provides fast IPC, and a chief-and-clan mechanism [16] to realize message redirection, on top of the hardware.

As mentioned in Section 6.4 this layer also provides services to manage access to hardware resources. These are currently the resource manager RMGR, which comes with the Fiasco distribution and controls access to interrupts, manages capabilities to start new tasks and acts as a (currently non-persistent) root pager. Further services under development are services to manage DMA channels and to access I/O ports.

7.2 Access Control

Although currently no system-wide policy enforcement is implemented, this subsection will describe how message redirection, as presented in Section 6.2, could be implemented using clans and chiefs.

To keep access control flexible we divide it into a policy-independent access control enforcement facility (ACEF) and a policy-dependent decision facility (ACDF), as outlined in Figure 4, similar to object managers presented in [23]. The ACEF, the application's parent task called chief, intercepts all messages sent by one of its child tasks (clan) to check whether the IPC message is permitted. Depending on the security policy, the ACEF is

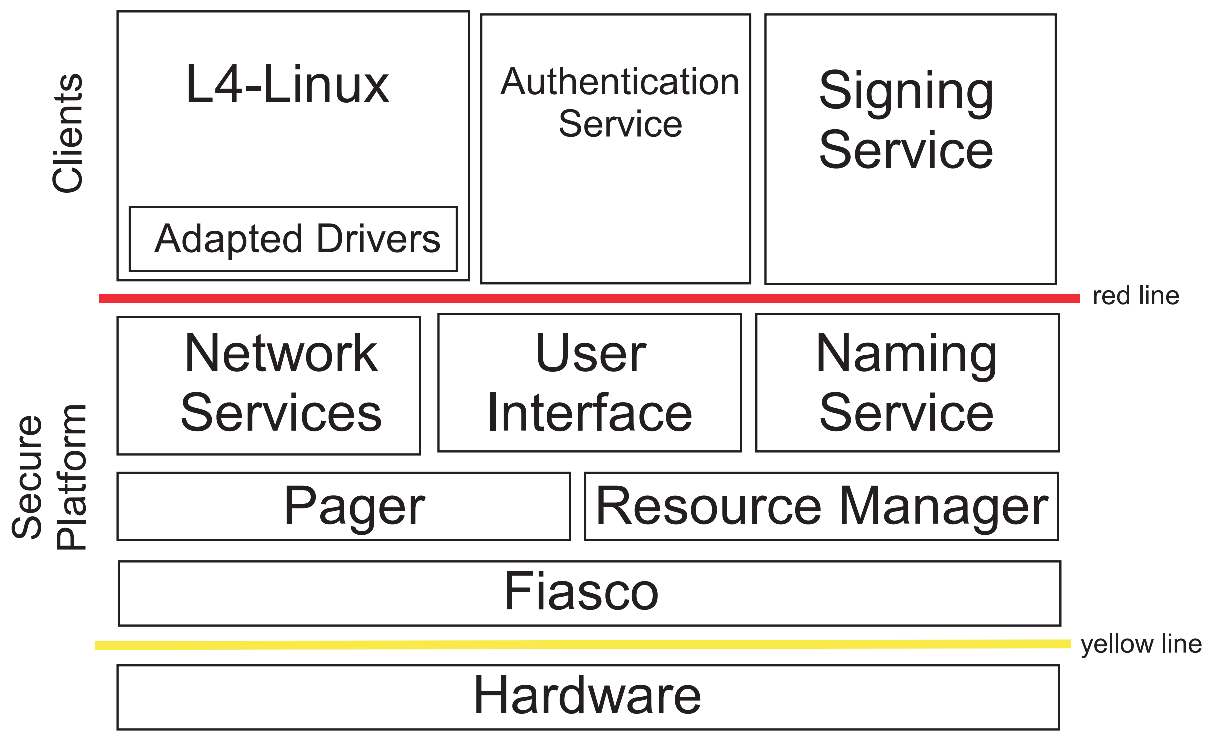


Figure 3: Currently implemented Perseus components.

able to enforce a system-wide policy by forwarding the IPC to the system-wide ACDF, or it can enforce its own policy by accessing its own ACDF.

The security policy also decides whether every application has to be protected by its own ACEF or whether one ACEF controls several applications (compartment). Because the decision logic can be defined by software rules within the ACDF we are able to support a wide range of security policies. To ensure that components outside the secure platform cannot bypass the ACEF, it controls its own interface and protects the ACDF.

7.3 Public Interfaces

Three servers providing interfaces available to client subsystems are currently implemented [24]. These are a naming service that converts interface names into thread IDs and vice versa, and a user interface service that manages access to the keyboard controller and the console. It guarantees that no task except the user interface is able to access the topmost line of the console, which is used to display the task ID and the given name of the application (see Figure 5). Currently only two compartments, trusted and untrusted, are supported. The μ-kernel , resourcemanager user authentication and signature service are trusted, and Linux and its subtasks are untrusted. Because data sent by the keyboard controller is only forwarded to the client that controls the remaining part of the display, we can prevent untrusted clients from spying security-relevant keystrokes such as pass phrases.

The third service implemented is a router service which provides network support by routing TCP/IP packets between Linux and trusted clients.

7.4 Clients

One client that runs on top of the secure platform and provides a common ABI is L4-Linux, a Linux kernel running on top of the L4 interface [8, 9]. Its device drivers are modified in such a way that they use the public interfaces presented in the preceeding section.

An authentication service (Figure 6) that controls the user interface after the boot procedure enforces the Perseus authentication mechanism. If authentication has been successful

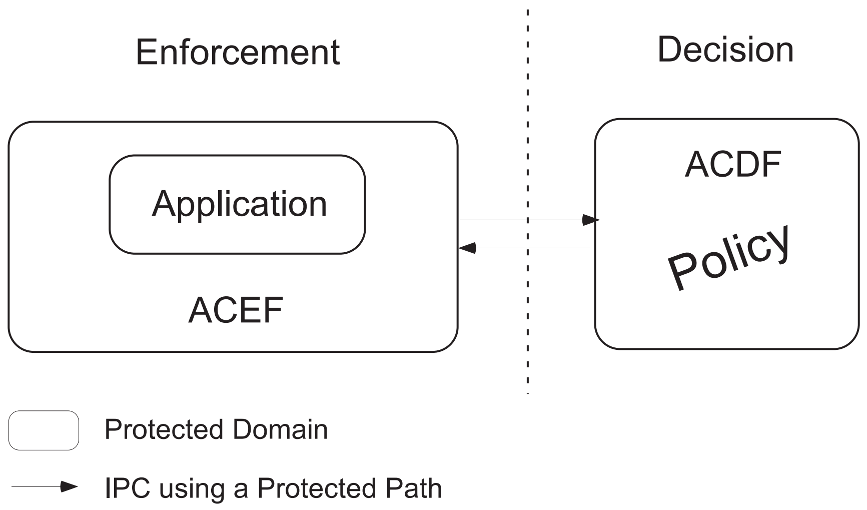
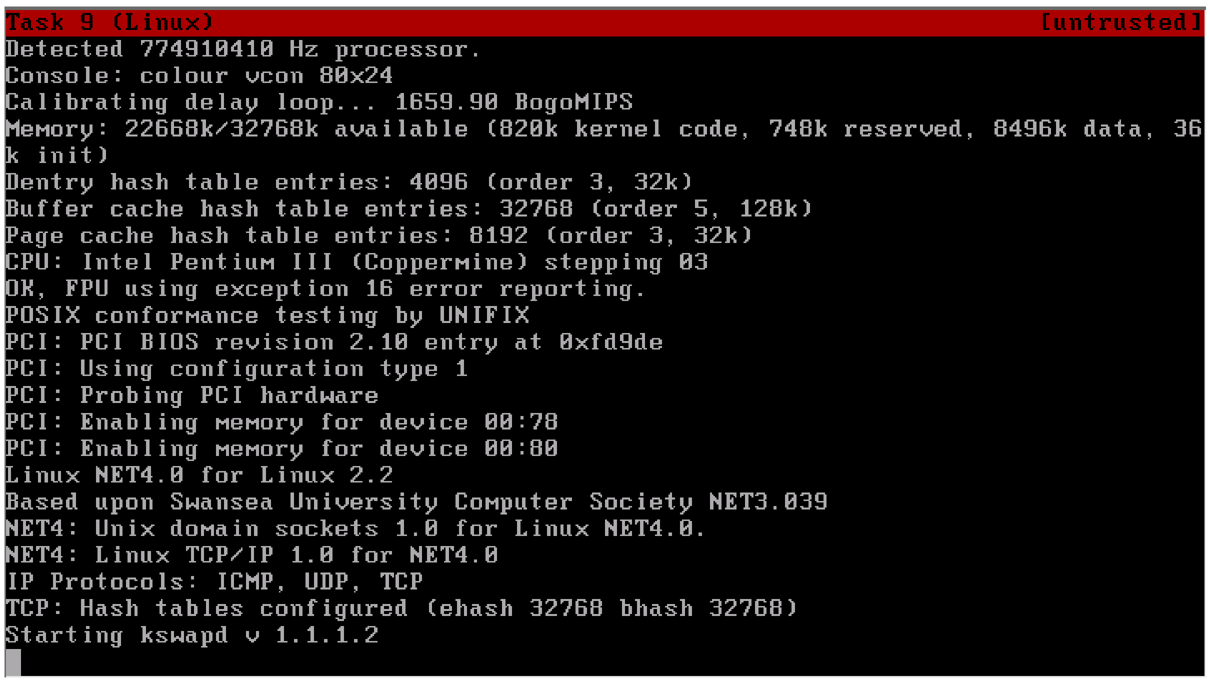


Figure 4: We divide the access control mechanism into an policy-independent enforcement facility (ACEF) and a decision facility (ACDF).



```
Task 9 (Linux)                                                      [untrusted]
Detected 774910410 Hz processor.
Console: colour vcon 80x24
Calibrating delay loop... 1659.90 BogoMIPS
Memory: 22668k/32768k available (820k kernel code, 748k reserved, 8496k data, 36
k init)
Dentry hash table entries: 4096 (order 3, 32k)
Buffer cache hash table entries: 32768 (order 5, 128k)
Page cache hash table entries: 8192 (order 3, 32k)
CPU: Intel Pentium III (Coppermine) stepping 03
OK, FPU using exception 16 error reporting.
POSIX conformance testing by UNIFIX
PCI: PCI BIOS revision 2.10 entry at 0xfd9de
PCI: Using configuration type 1
PCI: Probing PCI hardware
PCI: Enabling memory for device 00:78
PCI: Enabling memory for device 00:80
Linux NET4.0 for Linux 2.2
Based upon Swansea University Computer Society NET3.039
NET4: Unix domain sockets 1.0 for Linux NET4.0.
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP
TCP: Hash tables configured (ehash 32768 bhash 32768)
Starting kswapd v 1.1.1.2
```

Figure 5: The topmost line (red) under control of the secure kernel provides information about the task (in this case Linux) which controls the rest of the console.

the authentication service transfers control of the console to the Linux kernel, which is now able to enforce its own authentication.

The third client is a key-generation and signing service that can be accessed by a wrapper application available under L4-Linux to generate new key pairs and to sign documents using these keys. Because the key pair generated is stored within the signing service's address space, the document to be signed is copied into this address space and displayed via the trusted ASCII editor to allow user verification. If the user agrees, the document is signed and copied back into the Linux application's address space. Given that no access control mechanism is implemented so far, secure applications have to enforce their own security policy by controlling incoming IPC messages themselves. This is done by the signing service using password-based protection. Whenever a new key is created, the service requests the user to enter a password which has to be re-entered whenever a signature is being created. Of course neither the keystrokes nor the password itself can be accessed by the L4-Linux kernel and its applications.

8 Conclusion

We have presented a system architecture of a secure operating system that takes advantage of the entire range of available Linux applications. Nevertheless its minimalistic security kernel provides a high level of assurance because of its clarity. During all development steps both desktop systems and mobile devices have been considered, and its flexibility allows the system to be adapted to end-user requirements. We separate access control mechanism and decision logic to be able to support an entire range of security policies, and the modular concept of the system architecture allows changes of system services on demand. Because the client OS's device drivers are replaced by drivers that directly use the services of the secure platform, there is no need to provide a complete virtual hardware layer and therefore we expect only a small loss of efficiency.

Although the first prototype presented offers an environment for testing new services, considerable work still has to be done until a secure end-user system can be provided. We are optimistic that we will be able to evaluate the security kernel or even formally prove the correctness of the implementation.

References

[1] James Adams. The future of war. *Business Briefing: Global Info Security*, 2000.

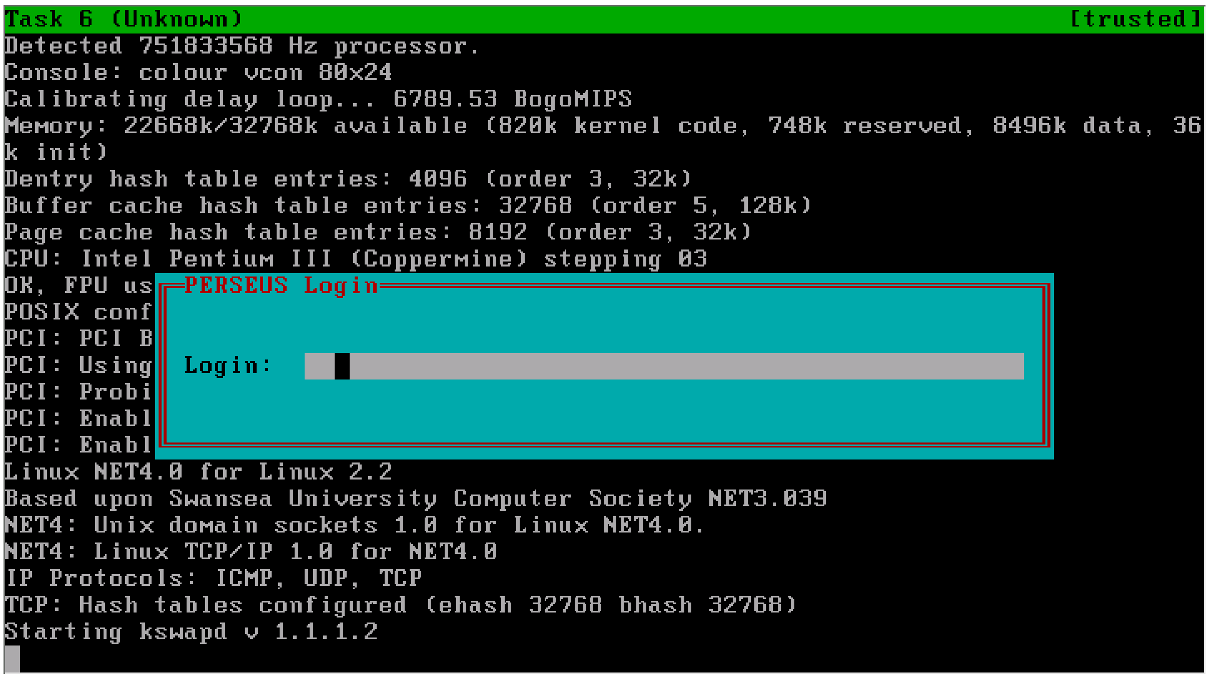
[2] P. Baur, T. Plasa, P. Kejwal, R. Drexler, W. Reif, W. Stephan, A. Wolpers, D. Hutter, C. Sengler, and E. Canver. The verification support environment VSE. In *IFA Symposium on Safety, Security and Reliability of Computers*, 1992.

[3] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Pronciples (SOSP-15)*, pages 267–284, 1996.

[4] F. J. Corbato and V. A. Vyssotsky. Introduction and overview of the multics system. In *Proceedings of 27. AFIPS Joint Computer Conference*, pages 619–628, 1965.

[5] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques (WIFT'95)*, June 1995.

[6] M. Gasser. *Building a Secure Operating System.* Van Nostrand Reinold Company, New York, 1988.



```
Task 6 (Unknown)                                                      [trusted]
Detected 751833568 Hz processor.
Console: colour vcon 80x24
Calibrating delay loop... 6789.53 BogoMIPS
Memory: 22668k/32768k available (820k kernel code, 748k reserved, 8496k data, 36
k init)
Dentry hash table entries: 4096 (order 3, 32k)
Buffer cache hash table entries: 32768 (order 5, 128k)
Page cache hash table entries: 8192 (order 3, 32k)
CPU: Intel Pentium III (Coppermine) stepping 03
OK, FPU us┌PERSEUS Login──────────────────────────────────────────────┐
POSIX conf│                                                           │
PCI: PCI B│                                                           │
PCI: Using│ Login:                                                    │
PCI: Probi│                                                           │
PCI: Enabl│                                                           │
PCI: Enabl└───────────────────────────────────────────────────────────┘
Linux NET4.0 for Linux 2.2
Based upon Swansea University Computer Society NET3.039
NET4: Unix domain sockets 1.0 for Linux NET4.0.
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP
TCP: Hash tables configured (ehash 32768 bhash 32768)
Starting kswapd v 1.1.1.2
```

Figure 6: The Perseus user authentication service controls the display. The color of the topmost line has changed to green (trusted) and the service shows the login dialog.

[7] A. Gefflaut, T. Jaeger, Y. Park, J. Liedke, Kevin J. Elphistone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuter. The SawMill multiserver approach. ACM SIGOPS European Workshop, September 2000.

[8] H. Härtig, M. Hohmuth, and J. Wolter. Taming Linux. Proc. 5th Australasian Conference on Parallel and Real-Time Systems (PART), Adelaide, September 1998.

[9] M. Hohmuth. Linux-Emulation auf einem Mikrokern. Diplomarbeit, Dresden University of Technology, Dept. of Computer Science, August 1996.

[10] M. Hohmuth. The Fiasco kernel, requirements definition. Technical Report ISSN 1430-211X, Dresden University of Technology, Dept. of Computer Science, December 1998.

[11] H. Härtig, O. Kowalski, and W. E. Kühnhauser. The BirliX Security Architecture. *Journal of Computer Security*, 2(1):5–21, 1993.

[12] C. Jacobi and D. Kröning. Proving the correctness of a complete microprocessor. In Kurt Mehlhorn and Gregor Snelting, editors, *Informatik 2000, Proc. 30. Jahrestagung der Gesellschaft für Informatik*. Springer-Verlag, 2000.

[13] T. Jaeger, K. Elphinstone, J. Liedke, V. Panteleenko, and Y. Park. Flexible access control using IPC redirection. *IEEE HotOS*, 3, 1999.

[14] T. Jaeger, A. Prakash, J. Liedke, and N. Islam. Flexible control of downloaded executable content. In *Transactions on Information and System Security*, volume 5, pages 177–228. ACM, May 1999.

[15] J. Liedke. *L4 Reference Manual*. GMD/IBM Watson Technical Report, 1996.

[16] Jochen Liedke, editor. *Clans and Chiefs. Architektur von Rechnersystemen*, 12. GI/ITG Fachtagung, Kiel, 1992.

[17] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrel. The inevitability of failure: The flawed assumption of security in modern computing environments. Technical report, National Security Agency, 1998.

[18] Common Criteria Project Sponsoring Organisations. *Common Criteria for Information Technology Security Evaluation (Version 2.1).* Common Criteria Project Sponsoring Organisations, 1999. adopted by ISO/IEC as Draft International Standard IS 15408 1-3.

[19] A. Ott. Rule Set Based Access Control as proposed in the 'Generalized Framework for Access Control' approach in Linux. Master's thesis, Universität Hamburg, Germany, 1997.

[20] A. Pfitzmann, B. Pfitzmann, M. Schunter, and M. Waidner. Trusting mobile user devices and security modules. *IEEE Computer*, 30(2):61–68, February 1997.

[21] J. S. Shapiro. *EROS: A Capability System.* PhD thesis, University of Pensylvania, USA, April 1999.

[22] J. S. Shapiro and S. Weber. Verifying the EROS confinement mechanism. In *Proceedings of the IEEE Symposium on Research in Security and Privacy 2000*, pages 166–176.

[23] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *Proceedings of the Eighth USENIX Security Sympositum*, pages 123–139, August 1999.

[24] C. Stüble. Development of a prototype of a security platform for mobile devices. Master's thesis, University of Dortmund, 2000.

[25] H. Tews. Case study in coalgebraic specification: Memory management in the Fiasco microkernel. Technical Report TPG2/1/2000, Inst. Theor. Informatik, TU Dresden, May 2000.

[26] P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, and G. Back. Formal methods: A practical tool for OS implementors. In *Proceedings of the 6th IEEE Workshop on Hot Topics in Operating Systems*, 1997.

[27] J. D. Tygar and A. Whitten. WWW electronic commerce and java trojan horses. In *2nd USENIX Workshop on Electronic Commerce*, pages 243–250, 1996.

[28] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM 17/6*, pages 337–345, 1974.