Security Architectures Revisited

Hermann Härtig

Technische Universität Dresden
haertig@os.inf.tu-dresden.de

Abstract

The knowledge in technologies needed to build secure platforms, or Security Architectures, *has significantly matured over the recent years. These include small interface technologies, access-control contracts, tunneling, secure booting, effective resource control, and virtual machines. Putting together these ingredients into a small secure platform seems straightforward, yet still remains to be done, and has the potential of making operating systems more dependable.*

1 Introduction

In the last decade, several (operating) system projects were well underway to build platforms for applications with very high security requirements. Examples include DSSA (Digital Systems Security Architecture [5]), Trusted Mach [2], and BSA (BirliX Security Architecture [7]). None of them was used in practice, at least not in notably large scale.

One reason may be the (right or wrong) perception that such systems were and — despite the events on September 11th — are not needed. Another reason may be that the technology to build such systems was not mature enough at the time.

This paper claims that today the knowledge to build such a platform is well established in the operating-systems community and that the required technologies have significantly matured over the recent years. However, they still need to be combined into a proper architecture. The remainder of the paper shortly states my view of the requirements for a small secure platform and discusses the needed technologies and their status. It finally puts these ingredients together into a general-purpose small secure platform and explains the use in a dedicated embedded system.

2 Requirements

Devices such as mobile phones or PDAs, but also notebooks and desktops will be and actually are used for valuable or critical applications such as banking transactions, while on the same device all sorts of potentially dangerous rubbish applications are downloaded from the Internet. Embedded systems are increasingly connected to and controlled via the Internet. In comparison to such systems, servers have advantages since they are (at least should be) in a physically controlled and carefully administrated environment. But also for servers, mobile code — intended as such or unintended — presents a severe threat. The prime requirement for these systems is that the valuable components and applications are reliably protected from the other parts.

To achieve this protection, a careful implementation of a small secure platform (to avoid the infected term of a trusted computing base) is needed that

- provides minimal yet sufficient functionality for applications with high security requirements,
- is flexible enough to be used either as complete general purpose platform, as a scaled down dedicated embedded system, or as a thin server,
- enables in real practice the employment of the *principle of least privilege*, especially but not only for mobile code, and uses it in the architecture,
- supports separation of secure and insecure parts of the system to an extent that even the successful penetration of the core of the insecure part does not endanger the secure side,
- can prove its identity to near or far away communication partners,
- provides compatibility for legacy applications, is an open (in contrast to a closed language) system, and supports reuse of potentially insecure components (e. g., of a network protocol stack), and
- is still small and simple enough for thorough evaluation.

Small and simple enough in more precise terms is in my view: A small group of people, for instance around seven, must be able to completely control the secure platform, i. e., each member of the group should understand all interfaces of the components of the architecture, and each component should be completely understood by at least one member of the group. Systems of the size of the Linux kernel, not to speak about recent versions of other desktop operating systems, will have tough times to achieve this property.

3 Technologies

The technologies needed to build a small secure platform are small-interfaces technologies, tunneling, secure booting, access-control contracts, effective resource control, and virtual machines. We discuss each shortly and point out advances and limitations of their current state of the art.

3.1 Small-interface technologies

For a secure platform to be under complete control of a small group, it must be built as a small set of small components with small interfaces. An example for a large interface is interaction of components by unrestricted usage of pointers in a shared address space, still the most favorite way to build large, monolithic operating-system kernels.

In recent years, the operating-systems community developed two (competing) technologies to build systems with small interfaces. Both are based on small kernels that are supposed to be small enough for thorough evaluation. Then systems are extended either by adding extra functionality on top of these kernels and in their own address spaces or by downloading extra functionality in safe ways into these kernels. The former is usually referred to as microkernel-based approach while the latter is called the extensible-systems approach.

In the **microkernel-based approach**, small interfaces are enforced by separate address spaces, which are effective even in the presence of pointers. Unsafe components can be isolated in their own address spaces. The long held perception of microkernel-based systems to be extremely slow has been proven wrong by work based on the L4 family of kernels [11, 6]. However, the applicability of these results to systems with really small interfaces still needs to be investigated because these current systems use large components (single-server Linux) and make use of shared regions of memory.

A difficult problem of the microkernel approach comes with input-output drivers, if unrestricted use of DMA is allowed. Then, a malicious component, though encapsulated in its own address space, may initiate DMA transfers to any physical address, thus in fact breaking the encapsulation via address spaces. We know of two approaches to tackle that problem:

The first one is to disallow DMA for untrusted components. One technique to do that is to rely on virtualization of hardware which in practice can be done for a limited number of devices only. The other technique is to mediate all DMA accesses through a trusted server. Again, current PC hardware with its not-exactly-systematic DMA interfaces makes it necessary to look at each driver separately. The situation becomes even harder for programmable devices.

The second approach is to restrict DMA access to a partition of physical memory by hardware. This is simple if a bus controller supports this. Then, all DMA, including that from trusted components, first goes into that partition and then is copied to the trusted component's address spaces. The content has to be protected by other means (see next subsection on tunneling techniques). To solve this problem more thoroughly, additional help may be needed from hardware designs by providing a notion of address spaces on devices or bus controllers (sometimes referred to as DVMA). This may come with as little effort as a riser card for PCI devices [17].

In the **extensible-systems approach**, small interfaces are enforced by restricting the components to be downloaded into the kernel. A common restriction is the use of safe languages [3], which implies inherent difficulties in dealing with input-output drivers. Another approach is to include a transaction-like mechanism into the kernel [15]. However, this seems to cause even harder performance problems than those of the microkernel fraction. A third line to mention is having pieces of code carrying their own proofs which are checked before activation [12].

Both technologies are developed far beyond the status of the ones available to the builders of the previous attempts to build security architectures. As an example, the Mach kernel used as base of the Trusted-Mach architecture, was more than ten times larger (interface, size, cycles, ...) than L4.

In addition to supporting small interfaces, both technologies have the potential to provide legacy operating-system interfaces to support legacy applications, a requirement listed in Section 2. For instance, to move the Linux kernel onto user level as a binary-compatible single-server emulation of the Linux system-call interface required changes respectively additions of around 7000 lines of code. Performance comparisons between a Mach-based and an L4-based single-server implementation of the Linux kernel bring home the point that the technology has matured significantly [6].

3.2 Tunneling

Tunneling is a technique to use software that by itself does not provide a required property and adding this property in an additional layer. Well-known examples are using an insecure protocol for secure communication by encrypting packets before handing them over or tunneling IPv6 packets over IPv4 channels. The term tunneling as used in this paper may be overextending the more common uses of this term.

Potential applications of tunneling for a small secure platform include the file system and input-output drivers:

The **file system** does not need to be part of a small secure platform unless denial of service is a concern. Instead, an untrusted file system can be used to store encrypted information. This implies for a security architecture that an existing file system can be reused, for example the file sys-

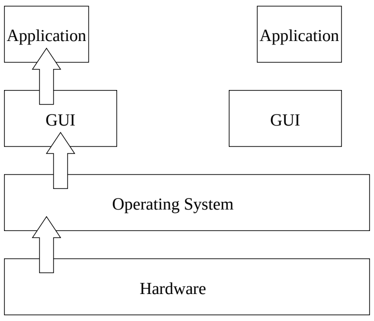

Figure 1. Authentication chain

tem of a complete Linux system in the insecure part of the architecture. Of course, an adversary can destroy data in the Linux file system after a successful intrusion, but can neither obtain access to confidential information nor apply unnoticed changes. This may be completely acceptable if a local file system merely acts as cache for a remote but accessible file server.

Still, the ability to protect some storage such as a minimal flash file system for cryptographic keys needs to be part of the minimal secure platform. If enough such protected storage is available, it may make sense as well to store data that has been modified since last backup, which turns the file system into more than just a cache.

Input-output drivers increasingly become the most complex, ugly, and least-controlled parts of systems. Their partial removal from the trusted part would enormously increase the possibilities for a thorough evaluation of a platform. Though nobody tried this, it seems possible, if and only if DMA management can be put under control. For instance, a trusted file system does not need a trusted disk driver.

The inherent limitation of tunneling however is in denial of service attacks. If a driver is needed for a requested functionality, it must be part of the secure platform. But in embedded appliances, where Internet access is often needed for reporting purposes only, protocols can be left out of the secure platform and reused using tunneling techniques. Hence, an aggressive combination of small interface technologies and tunneling indeed promises to have the potential of keeping the small platform small enough to stay under complete control of a small group of developers.

3.3 Secure booting

Neither microkernels nor tunneling solve the problem of What You See Is What You Get. It is easy to emulate a user interface or a complete device that pretends to be something which it is not, hence prompting the user to unveil secrets involuntarily. A recent successful attack on PGP made use of exactly this phenomenon.

The technique to solve this type of problem has been known for over 10 years [5, 7] as *secure booting.* Secure booting ensures that a specific hardware with a specific OS with a specific GUI and a specific application is indeed running in the identified device. Secure booting relies on hardware to establish the identity of a boot loader, on the boot loader of the operating system, on the OS of the user interface, and so on, thus forming a bottom-up chain of authentication rooted in hardware (Figure 1). An authentication protocol can then be used to verify this chain from a remote computer. A remote computer may be a server several thousands of miles away or a smart card used to locally identify a device.

Recent attacks on cell phones can be attributed to this class of problems. There, cheap cell phones were sold under the precondition that for a certain limit of time only a certain provider can be used. It was part of the operating systems' functionality to enforce this deal. The attackers stole the cell phones and replaced the operating systems. Their communication partners were not able to find out whether or not the cell phone was using its original operating system.

There are two limitations that need to be mentioned. One is the need for physical protection of the devices, notably their tamper resistance and freeness of side channels. Side channels are means to extract confidential data such as cryptographic keys using *unintended interfaces.* An example for an unintended interface is power consumption that changes depending on whether a 1 or a 0 is currently processed in a cryptographic key. An obvious unintended interface is direct access to the memory bus of a device, for instance with in-circuit emulators. Tamper resistance requires — in contrast to tamper proofness — that unnoticed modifications are impossible (or very unlikely). Both avoidance of unintended interfaces and tamper resistance are hard to achieve, depending on the efforts of the attacker. A constant flow of news from the University of Cambridge impressively brings home that point.

The other limitation against which secure booting does not defend is an attack that is sometimes referred to as the *Mafia Fraud.* Here, an adversary replaces a device with a faked one that forwards all communication to the original one and thus successfully performs all required protocol steps of the secure booting protocol. It then can show an arbitrary user interface that prompts the user to reveal secrets involuntarily. Several techniques to solve this problem have been proposed. The most promising in my opinion is based on frequent, key-controlled hopping between large numbers of channels. This solution will be presented at the upcoming military communication conference [1].

The apparent change within the past years is that manufacturers started building devices based on such ideas.

3.4 Access control based on contracts

In practice, access rights in end-user systems are granted much too lavishly. The main reason for that — besides negligence — is the complexity of mapping an intended security policy to discretionary access-right mechanisms, at least as provided by today's platforms. Rule-based schemes did not help in practice either, since the known sets of rules turned out to be of not much use outside of the domain of military document systems. In other words, an intuitive and practical way of correctly employing the principle of least privilege is missing. In the increasing presence of mobile code — some prophets are even foreseeing mobile code as the governing principle for software usage — this situation is dangerous.

Help may come from lifting the process of granting access rights up one level of abstraction. Access rights then are based on contracts of the following kind: *If I get access right to these specific resources, then I will perform that specific function for you.* The requested access rights have to be shown up front. For example, before a new program is installed, it needs to show the contract, i. e., which resources it needs to perform which function. Then, before an installed program is executed, it explicitly presents the access rights to the objects it needs beyond the access rights already granted to the installed program, if any. As another example, before a new piece of mobile code is downloaded and started, it needs to state its needs and promised function.

Several implementations of this scheme have been published. Even UNIX can be seen as having an early version of this: Manual pages used to contain a section "used files" stating the resources needed, yet were used for documentation only.[1] All of these implementations share most of the techniques and properties, but also at least one major hard problem that needs to solved:

Programs in general and mobile code in particular become offers for contracts, i. e., together with actual code and identification of vendor, each program contains a (symbolic) list of resources needed. To ensure this mapping of code to vendor and requested resources, the integrity of contracts must be enforced by cryptographic signatures. Whenever a program is installed, the list of requested resources is shown up front and offered to the user for approval.

Constantly being requested to approve contracts will certainly lead to the *too-many-alarms effect* and in consequence to unintended approvals of contracts. Hence, shortcuts are needed. One way to provide shortcuts is the use of (symbolic) access-control lists (ACLs) expressing trust with regard to sources of code. For example, all programs of a certain distribution or package are trusted for a specified set of objects. Whenever a new process is started, the requested rights are first checked against these ACLs before confronting the user.

The result of an agreed-upon contract is a (symbolic) list of resources the process (the program in execution) may use, in principle similar to a (classical and old-fashioned) capability list. Accessing an object, for instance opening a file, without possessing the proper capability is a violation of the contract. In contrast to classical capability and ACL schemes, capabilities and ACLs are in symbolic form, e. g., in some platform-independent form that can be mapped to system-dependent names, for instance in a Unix-like system to file or DNS names or to symbolic representations of parameters. These capabilities then can and must be inspected at all invocations that carry symbolic names, for instance in a Unix-like system at open, connect and bind, and various exec system calls. Classical capabilities containing non-symbolic identifiers of object and access rights to them, for instance Unix's file descriptors, are created by some of these invocations and are used and checked thereafter. Adding enforcement of symbolic capabilities to a system like Linux requires about 350 lines of changes to the kernel [10].

Capabilities, at least at the symbolic level, may have to include parameters. For instance, to control access to a modem, it makes sense to include the requested telephone numbers. Again, this is simple if a symbolic command-line interface such as often used to invoke Unix programs is enforced for accesses to resources in question.

An example for a symbolic requested resource is *"dial 1 800 123 4567891"* which is accepted if an ACL contains *"dial 1 800 *"* . Sometimes, mappings are expressed explicitly, for instance a program may request access to *"mailer-spool-dir"* and the corresponding ACL may contain *"mailer-spool-dir is /var/spool/mail"*.

From our experience, the **hard problem** in practice is that all interpreters must be made to obey access-control contracts. For example, a JVM running in a network browser may either implement it by itself[2] or the underlying operating system may force it to do so. The former requires trust in all interpreters, and there are (too) many, as the spread of macro viruses via document editors shows. The latter requires the interpreters to be modified such that for each new contract (e. g., a downloaded applet) a new encapsulated entity of the underlying system (e. g., a new process running the browser in Unix-like systems) needs to be created. The contract, the resources needed by the new browser process on behalf of the newly downloaded applet, needs to be forwarded to the underlying operating system and attached to the newly created encapsulated entity (Unix process). This problem becomes more complicated for more interesting forms of interpreter nesting. The attempts of several of my bravest students to do so with the Netscape browser failed miserably.

[1]This analogy was pointed out by Sape Mullendar when explaining how the Amoeba architects intended to generate capabilities from machine readable documentation.

[2]IBM's Flexxguard System implements a similar scheme.

3.5 Effective resource control

Effective defense against denial-of-service attacks through blocking of resources needs effective resource control. Work in the real-time systems community has brought significant progress in two areas.

One of them allows better control over the allocation of resources to parts of a system. Systems such as Resource Kernels [14] and DROPS [8] in principle allow reserving resources independent of their type, i. e., CPU as well as disk bandwidth, for example. While, to my knowledge, such systems are still very much in prototypical status, their potential is promising.

The other area is a technique called early demultiplexing. While in classical implementations of protocols, the decision on whether or not to throw away a packet came rather late and thus wasted resources just for policing, newer implementations even making use of small dedicated CPUs in off-the-shelf network-interface cards have proven the progress. Examples with numbers are given in [4].

Both techniques seem helpful in preventing localized denial of service attacks based on intentional resource exhaustion. Dealing with distributed attacks will require support from routers or lawyers.

3.6 Virtual machines

Virtual machines support legacy at an even lower level in comparison to the emulation of operating system interfaces. Legacy operating systems run (nearly) unchanged. This replaces the *n*-fold effort to emulate *n* legacy operating-system interfaces by the emulation of just one hardware architecture. This technique, pioneered by IBM's mainframes, became available for PC architectures in recent years.

Isolation or separation is supported by providing different machines for different (classes of) applications. Especially, it solves the DMA problem mentioned in Section 3.1 by emulating input-output devices such that even malicious drivers cannot break the separation (as is possible in current small-interface technologies). However, this comes at the cost of emulating the devices, which is higher than the cost involved with identifying and controlling their DMA accesses.

3.7 More on technologies

This paper concentrates on technologies in the operating-systems domain. However, it must be stated that software-engineering technologies to systematically build more reliable software in general have matured as well since the last significant efforts to built security architectures.

Methods for static analysis to discover certain types of faults have advanced. C (and C++), the most favored language(s) for operating-systems builders, have never earned

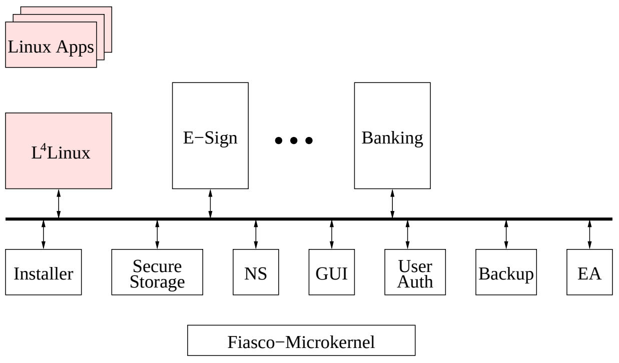

Figure 2. The Nizza architecture

the reputation of being the most advanced language with respect to safe programming. Now, some restricted versions of C are proposed that maintain its versatility, but allow stricter analysis.

These and related technologies of good software engineering and programming practice are considered as orthogonal to the architectural technologies discussed in this paper. I ignore them henceforth.

4 Architectures

Putting together these technologies into a security architecture based on a small secure platform seems rather straightforward in principle. Still, this section should be taken with a ton of salt, since so far it is educated speculation to a very large extend. We sketch a rather general-purpose platform (Nizza) and an application of the technologies to a dedicated system (Micro-Sina).

4.1 Nizza

Nizza[3] is a back-of-the-envelope (literally speaking) design for a small secure and general-purpose platform supporting applications with high security requirements such as digital signatures and banking protocols while still having the option of running legacy code. It is sketched in Figure 2. Discussing Nizza serves as an opportunity to identify the critical components and the role of technologies discussed before.

The architecture separates the system into two parts, the legacy and potentially rubbish part (left) and the secure part. Both, legacy and secure sides, run on the small secure platform whose principle tasks are to provide minimal sufficient functionality for applications with high security requirements and to ensure separation. Applications on the secure side reside in their own address spaces. They contain all functionality which is needed but not provided by the small secure platform, e. g., as libraries. New applications on the secure side can be installed from outside by

[3]Nizza is the German name of Nice, France

contacting the legacy side which forwards such requests to the installer. This also is the first usage of tunneling: Rather than providing transport and higher-level protocols in the secure side, the insecure side's functionalities are used.

The unsafe side, L^{4}Linux and its applications, is based on a user level implementation of the Linux kernel [6]. It uses transparent libraries as invented by Mach for binary compatibility for most applications. L^{4}Linux is running in its own set of address spaces.

The prime requirement for the insecure side is that it cannot harm the secure side, even if its core, the former *Linux Kernel*, is penetrated successfully. L^{4}Linux — on an as-it-is basis — needs to be *tamed* in several aspects:

- L^{4}Linux needs to be modified to have the X-Windows server run on a frame-buffer implementation that is provided by a secure GUI (see below).
- Input-output drivers that need to be part of the small secure platform need to be taken out from L^{4}Linux, however can still be used via stubs from L^{4}Linux. This technique is reasonably well understood due to the reuse experience of drivers in the DROPS real-time system.
- DMA needs to be put under control to enforce address-space separation. If hardware support is available to restrict DMA to a partition of physical memory, all DMA — including the one initiated on the secure side — is directed to that partition and then copied to secure memory. Hence the contents must be protected by other means, e. g., via tunneling techniques. The effort needed for that and the resulting performance remain still unclear.

Having a virtual machine instead of L^{4}Linux would be desirable since it is more general with regard to legacy software. Then however, protocols to cross machines boundaries (e. g., TCP/IP) belong to the secure part.

The platform is based on Fiasco, a careful reimplementation of the L4 interface. It provides address-space separation as a basic means to keep interfaces small. The L4 microkernel interface has about a dozen system calls, hence the interface can be considered small. It is not only under complete control of one person, but is small enough to have triggered Dresden's theory group to leave alone their stacks and lists and try a source-code-based formal verification of a claim that is fundamental for address space separation: "Only kernel-code runs in kernel mode" [9]. The major drawback of the current situation is L4's elegant but notoriously inflexible chiefs&clans mechanism [11]. Considerations are well under way to replace it by somewhat else in one of the future versions of the interface.

An important component of the small secure platform is the name server. It provides a symbolic name interface to all resources of the small secure platform including a communication interface to the insecure side. It maintains the certificates needed to check the validity of requests from outside. An important role of the name service is to provide symbolic names that are used in access-control contracts, capabilities, and ACLs.

The "installer" is the component responsible for loading and installing other components of and new applications on the secure side of the system. The installer's responsibility includes the decisions on newly-proposed access-control contracts and the derivation of symbolic access-control and capability lists. The installer however needs to take care only of the small secure platform as an *interpreter*, not of nested interpreters. An installer is a complex component, alone due to the machinery needed to load and establish applications besides and independent of L^{4}Linux. However, it can rely on L^{4}Linux to load programs from servers. The installer needs to be aware of secure booting. It establishes the authentication chain and makes it available to the trusted GUI. We will subsume the boot loader and related components to be part of the installer and will not discuss that part of secure booting henceforth.

The secure storage component, if it is designed for confidentiality and integrity, again can rely on tunneling. The actual storage of large data can be left to the L^{4}Linux file system. Encryption technology can be used to protect data against information dissemination and unnoticed modifications. It cannot protect against destruction of data, i. e., against a denial of service class attack. However, this may be tolerable if almost all data on a PDA or cell phone is stored on a server anyway reducing the secure storage component to a mere cache for the largest part of data. However, direct, i. e., untunneled secure storage is needed for the keys and for data added since the last backup. It seems, that soon modern chips can have enough memory on chip to avoid having to add an extra external module to the tamper-resistant device.

A trusted GUI component must reliably show the authentication chain of the application that is currently controlling the screen. To this end, it needs to provide a interface to the X window system that disallows direct access to the video memory. The component to authenticate human users need to be part of the small secure platform as well. Keeping input-output drivers in the secure platform small, for instance by reusing existing drivers with tunneling techniques, will be a major challenge.

We assume that everything else can be done at the application level. This includes cryptographic infrastructures as needed by specific applications.

4.2 Micro-Sina

Micro-Sina[4] is an effort to replace a Linux-based implementation of a VPN box by one based on the Fiasco

[4]Micro-Sina is sponsored by the BMWi and done in cooperation with Secunet AG.

microkernel. The objective is to identify and implement the minimal functionality that is needed for that purpose. Micro-Sina will (probably) not include secure booting since physical protection is provided for these boxes. It will contain a secure storage component, a name server, and some input-output drivers. In notable contrast to Nizza, L^{4}Linux can (probably) not be used for TCP/IP- or IPsec-tunneling. Reusing L^{4}Linux's IPsec implementation would require trust in L^{4}Linux, and reusing its TCP/IP implementation after having done the encryptions would violate IPsec's data-format obligations. Hence, according to our current understanding, we need to extract Linux's (or another system's) IPsec implementation and carefully port or completely rebuilt it from scratch on top Fiasco as part of the small secure platform. This is an example for a major limitation of the applicability of tunneling.

5 Related work

For related work, we will concentrate on security architectures, i. e., on the integrative usage of the key technologies rather than adding more references to the technologies per se. We will skip closed-language-based systems and real-time systems that employ separation techniques in similar ways (such as Oncore System's and DROPS' real-time variants of Linux). In my view, IBM's and VMWare's implementations of virtual machines and completely new implementations such as the EROS operating system come closest to what I claimed to be desirable. However none of these comes anywhere near to some derivatives of MULTICS and to the very well-written descriptions of the DSSA. The most complete integrative implementation of a Nizza-like architecture is probably Christian Stüble's Perseus [13].

5.1 Classical security architectures

DSSA [5] is centered around an elaborate authentication scheme which is rooted in hardware and extends up to the authentication of application processes. It is used in combination with very flexible ACLs that allow fine-grained authorization. Although DSSA has never been implemented completely, the authors claim credibly that they did not encounter difficulties that they suspected to be unsurmountable. DSSA was accompanied by a technique that allowed formal reasoning about the validity of claims of their protocols. Though these formalisms met scepticism in the cryptography community, it at least turned out a powerful tool in convincing peers about certain claims (as I had experienced once when being the object to a such as exercise). DSSA did not look at other techniques discussed in this paper. It did not care about the size of implementations of secure platform. Although the ACL scheme was fine-grained, there were no considerations about how to use them to express users' needs.

Trusted Mach [2] (T-Mach) is a careful implementation of multi-level security. It is based on a kernel derived from the Mach microkernel that implements a reference monitor for a multi-level security policy. T-Mach certainly suffered from the then state of the art in building small kernels. A fair appreciation of T-Mach based on the available information and the space available for this paper is not possible.

BirliX was a fairly complete, object-based reimplementation of the Unix kernel interface in a language safer than C. The effort needed to achieve true binary compatibility turned out enormous which leads to my perception that reuse of off-the-shelf operating systems for the insecure side is the method of choice. BirliX had reinvented secure booting but did not implement it either. It had a notion of combining capabilities (*subject restrictions* in BirliX speak) with ACLs of fine granularity. Subject restrictions were carried around by programs, coming close to but not quite arriving at access-control contracts.

5.2 Virtual-machine implementations

In discussions with strict believers in virtual-machine technology I observed the somewhat naive perception that virtual machines solve all problems radically per se.

They do not. First, separation of machines does not necessarily separate applications. It does not make a big difference whether network or local IPC is used to allow applications to communicate with the outside world. Some way of controlled interaction and installation of applications must be provided, e. g., access-control contracts. Second, the malicious-driver problem is solved only for drivers for emulated hardware devices, not for the drivers needed to implement the virtual machine. Hence, the small-platform problem that becomes hard when input-output devices are part of the platform still needs to be solved, best using some small-interface technology. Once a device is so well under control that it can be emulated, it is fairly easy to make sure the driver does not use DMA to corrupt other processes address spaces. Hence, the problem of providing encapsulated drivers remains hard. Third, virtual machines do not solve the *small* platform problem per se although building a small virtual-machine implementation seems possible. Fourth, the secure-booting problem is orthogonal to using virtual machines. Fifth, is does not help to provide virtual machines if the operating systems running on them are not secure.

However, some of the problems described in this paper are well addressed by some of today's virtual-machine systems. LPAR, IBM's implementation of virtual machines in their z-Series line of machines, is small enough to have earned an EAL-5 evaluation. Also, VMWare's new server-oriented implementations are not based on Linux or Windows 2000 anymore, but on a smaller kernel. Providing a virtual machine is certainly the technology of choice to support legacy.

Remains to say: *If only all CPUs would clearly separate user/kernel from metal/virtual issues to support efficient implementations of virtual machines (unlike the x86 architecture).*

5.3 EROS

EROS [16] — an operating system that has been developed from scratch — is based on a carefully designed capability system and persistent single-level storage. EROS addresses several problems mentioned in this paper.

EROS certainly possesses the basic security mechanisms to provide access-control contracts. Is does not (yet) address the problem of cooperating interpreters.

The authors claim that a Linux-compatible environment is in progress, but from our experience with BirliX we assume that this effort is in danger of becoming a never-ending hunt for a moving target. For legacy applications, we see no other practical chance than either virtual machines or adaptation and encapsulation of original operating-system implementations to a new underlying small secure platform.

For EROS, critical applications need to be refactored from existing ones into components to take advantage of the underlying kernel's security properties, which induces a significant development cost. EROS' authors do not seem to look in systematic application of tunneling techniques to save effort for the implementation of the secure partitions of the platform.

EROS postulates the implementation of distinguishable trusted and untrusted user interfaces but does not take into account adversaries that can replace EROS by another operating system. It does not address secure booting techniques to prevent that kind of attacks.

Overall, EROS, to our knowledge, is a well-designed capability system that addresses some but not all problems of this paper and uses some but by far not all technologies that are at hand for the design of secure architectures. As a side remark, we expect the anticipated effort to gain EAL 7 certification will be hard, since EROS is not to be based on a small-interface technology.

6 Acknowledgements

The envelope used for designing Nizza laid on a table on Nice's beach surrounded by Birgit Pfitzmann, James Riordan, Michael Waidner, Arnd Müller, and myself. Christian Stüble, PhD student at Saarbrücken, then started to undertake the brave attempt to bring the envelope to paper and to start implementing some of these ideas.

Many discussions with students and other members of the operating-systems and real-time group of Technische Universität Dresden helped a lot when writing this down. Michael Hohmuth and Christian Stüble have helped in finishing this paper.

Dresden's operating-systems and real-time group gratefully acknowledges generous grants from DFG, BMWi, IBM, Intel, and others that enabled this work.

7 Conclusion

The topic of this workshop is *Can we depend on OSes?*

The answer is: *We could much more so, if only the technologies that have significantly matured over the recent years would be put to proper use.*

References

[1] A. Alkassar and C. Stüble. Towards secure IFF - preventing mafia fraud attacks. Accepted for IEEE Military Communications Conference 2002 (MILCOM), Anaheim, California, Oct. 7-10, 2002.

[2] N. Associates. Trusted Mach — specifications. URL: `http://www.nai.com/research/nailabs/finished-projects/trusted-mach.asp`

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the Spin operating system. In *15th ACM Symposium on Operating System Principles (SOSP)*, pages 267–284, Dec. 1995.

[4] U. Dannowski and H. Härtig. Policing offloaded. In *Proceedings of the Sixth IEEE Real-Time Technology and Application Symposium*, Washington D.C., May 2000.

[5] M. Gasser, A. Goldstein, C. Kaufmann, and B. Lampson. The Digital distributed system security architecture. In *12th National Computer Security Conference (NIST/NCSC)*, pages 305–319, Baltimore, 1989.

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

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

[8] H. Härtig, L. Reuther, J. Wolter, M. Borriss, and T. Paul. Cooperating resource managers. In *Fifth IEEE Real-Time Technology and Applications Symposium (RTAS)*, Vancouver, Canada, June 1999.

[9] M. Hohmuth, H. Tews, and S. G. Stephens. Applying source-code verification to a microkernel — the VFiasco project. Technical Report TUD–FI02–03–März 2002, Dresden University of Technology, 2002. URL: `http://os.inf.tu-dresden.de/vfiasco/`

[10] S. Lehmann and A. Westfeld. Kapselung ausführbarer Binärdateien. slcaps: Implementierung von Capabilities für Linux. In D. Fox, M. Köhntopp, and A. Pfitzmann, editors, *Verlässliche IT-Systeme (VIS)*, pages 21–35. GI, Vieweg, Sep. 2001.

[11] J. Liedtke. Toward real µ-kernels. *Commun. ACM*, 39(9):70–77, Sept. 1996.

[12] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 229–243, 1996.

[13] B. Pfitzmann, J. Riordan, C. Stüble, M. Waidner, and A. Weber. The PERSEUS system architecture. Technical Report RZ 3335 (#93381), IBM Research Division, Zurich Laboratory, Apr. 2001.

[14] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for qos management. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, 1997.

[15] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 213–227, Seattle, WA, Oct. 1996.

[16] J. Shapiro and N. Hardy. EROS: A principle driven operating system from the ground up. *IEEE Software*, pages 26–33, Jan. 2002.

[17] R. Uhlig, R. Fishtein, O. Gershon, I. Hirsh, and H. Wang. SoftSDV: A pre-silicon software development environment for the IA-64 architecture. *Intel Technology Journal*, (4), 1999.