A Look at the EROS Operating System

Jonathan S. Shapiro
Johns Hopkins University

Abstract

EROS is a capability-based, secure operating system originally designed to address the needs of shared computing utilities involving mutually suspicious users. The deployment plan for the original design was to provide leased compute service to competing business entities, potentially running on a single machine. As a result, the system was structured to preserve security in the face of both hostile dynamic content and hostile users. While the EROS platform can support the efficient deployment of multilevel secure environments, it is primarily focused on *business* security requirements. These requirements often address integrity as well as security concerns. An EROS-based deployment can ensure that business-critical data is manipulated exclusively by authorized application software, which in turn is guarded against user tampering. Simultaneously, EROS enables users to run untrusted utility software, hand private or proprietary data to that software, and be assured (within reason) that the data cannot be disclosed to the outside world.

This paper gives a general overview of the EROS system and what it can do. The EROS research project has now ended. Further work based on the EROS system is being pursued under the CapROS project (www.capros.org). The EROS team has moved on to a successor system: Coyotos (www.coyotos.org).

1 Introduction

This paper accompanies a talk for the 2005 *Libre Software Meeting* in Dijon, France. Since it seems likely that this will be the last non-academic publication on the EROS system, it has emerged as a curiously informal mix of anecdote and paper. I wanted to provide not just a sense of what EROS can do, but also a sense of its history and rationale. In this paper I will try to describe how the EROS project got started, what is different and significant about the system, and why EROS or its successor might be interesting as a secure operating system platform for future applications.

EROS is a difficult system to get your head around, because it challenges many of the accepted wisdoms of conventional operating system design. Trying to approach this system by asking how it is like other systems that you may know is often a frustrating exercise. EROS is simply *different.*

One note to academic readers: there are some conclusions stated in this paper that are neither substantiated nor explained here. Some are addressed in other publications. Others result from chains of reasoning that are omitted for reasons of space. A few are opinions derived from years of in-depth experience with a range of operating systems. In an academic paper these statements would be inappropriate, but this paper is not aimed at an academic audience.

The EROS system is slowly taking on a life of its own. EROS is having significant influence on the successor to the current L4 system, which is on its way to being a pure, capability-based nucleus. The EROS code base itself has now been taken over by Charlie Landau under the CapROS project (www.capros.org). This seems fitting, since Charlie was one of the original KeyKOS architects. My own

This paper was first published as part of the *Libre Software Meeting*, Dijon, France, July 5-9, 2005

work has turned to a high-assurance successor system called Coyotos (www.coyotos.org).

As a research system, EROS has met essentially all of the goals that we set for it. Minor changes will be needed to move it into production use, but these do not appear to be especially difficult.

2 How the EROS Project Started

Like the LINUX system, EROS started its life as an attempt to provide an open source reverse engineering of an existing operating system: KeyKOS. Since there is a useful business lesson in it for other open source projects, I'll give a brief history here.

2.1 GNOSIS and KeyKOS

The GNOSIS project was started by Tymshare, Inc. in 1972. The system first ran commercial applications in 1983. For history fans, 1983 was also the year that UNIX System V and the 4.2 BSD UNIX systems released. The Tymshare system was named originally named GNOSIS ("Great New Operating System In the Sky"[1]), and the key technical contributors were Norm Hardy, Charlie Landau, Bill Frantz, Alan Bomberger, and Susan Rajunas. There is a paper describing the architecture of this system.[2] It is definitely a contender for the title of "densest piece of technical writing ever crafted in the history of computing."

In 1985, the GNOSIS system was spun out into a new company called Key Logic, Inc., and the system's name changed to KeyKOS. Key Logic was funded by Omron, Inc. to create a robust, secure operating system for the Luna 88K line of workstations. The technical effort was completed just in time for Omron to abandon the workstation business. Following this misfortune, Key Logic proved unable to make the transition to a world of commodity PCs, and ceased operations in 1990 after meeting every technical challenge they set out to address. As it was described to me at the time, they were "unable to make the necessary business transition from selling 7 copies of the system at $1 million each to selling 1 million copies at $7 each."

My first introduction to KeyKOS came through a talk given by Norm Hardy at Stanford University in 1989. Two years later I encountered it again as one of the founders of HaL Computer Systems. We briefly considered acquiring Key Logic, which had a strong engineering team and an extremely robust implementation. HaL's goal was to deliver a top to bottom line of systems that would be based on open software standards, but would provide the kinds of reliability that IBM customers had come to count on. To put some perspective on this, there is no UNIX-based system *today* (2005) that delivers the level of reliability routinely achieved by KeyKOS on less reliable hardware in 1990. As HaL's manager of software design and one of the founders, I was extensively involved in evaluating the KeyKOS system.

Two conclusions came out of that evaluation:

- Cary Liu, our manager for the operating system group, felt strongly that HaL would be better served by porting UNIX System V Release 4, a system that he had extensive experience with. Among other issues, Cary felt that he could hire developers to work on it who already knew the code base, and that this was critical to HaL's chances of success.
- I concluded that KeyKOS was far better than anything else I had seen, including SVR4. Which is kind of ironic in hindsight, because I had been the lead person in instigating the MIPS Application Binary Interface for SVR4 while at Silicon Graphics, and I had come to SGI from the language tools group at AT&T Bell Laboratories (home of SVR4).

I should add that Cary was absolutely right about this decision. Among other factors, HaL's global deployment partners were all committed to SVR4 as a software platform in some form, and transitioning them to another operating system, no matter how compelling, would probably have been impossible.

There are two business lessons here. The first is that managing a technology through a market dis

1 The names "GNOSIS" and "EROS" are a fairly convincing demonstration that technical people should not make important marketing decisions.

2 See http://www.cis.upenn.edu/~KeyKOS/, in particular *The KeyKOS Architecture*.

ruption (the introduction of the PC and the work station) is extremely difficult under the best of circumstances. In hindsight, it's clear that KeyKOS went after the wrong niche. You don't succeed in a technology business by taking on an entrenched competitor in an established market first.

The second lesson is that Key Logic made several critical mistakes in their financing. Tom O'Rourke, who had also been the founder and CEO at Tymshare, had learned how to start and run a company in a different generation, and I think he may not have realized just how much the world had changed and how cut-throat the industry would turn out to be during the 1990s. He needed multiple backers who were willing to play for high stakes and force him to articulate a viable business model. What he got was a single, conservative, corporate investor that decided the stakes were too high, and backed out of the entire industry just as it was becoming really interesting.

Anybody who is interested in these sorts of issues should definitely have a look at two books: Clay Christensen's *The Innovator's Dilemma,* and Geoffrey Moore's *Crossing the Chasm*.

2.2 EROS Version 0.1

After leaving HaL, I spent some time working with the KeyKOS team trying to find a new business opportunity for their system. By that point, Omron was fully out of the picture, but the remaining Key Logic investors were irreconcilably stuck. It quickly became clear that there was no way to obtain a viable license to the KeyKOS code base in any timely fashion.[3] Since I had never seen any of the KeyKOS code, or any non-public description of it, I decided that it couldn't be that hard to reverse engineer. Heck, I was an experienced kernel hacker, and how hard could it really be? Famous last words...

So I spent several days talking to Norm Hardy (the KeyKOS architect), walking through their documentation and asking him lots of questions. Norm wanted his technology out in the world for people to use, so we were both careful to avoid discussion of anything unpublished or proprietary. Which was how EROS really got started. After a year of part-time effort, EROS version 0.1 booted successfully on an i386 machine in early 1992, just in time to teach me about what happens when you have a total hard disk failure and no backups. I took a year out to recover by negotiating and managing the spin out of the Xanadu Operating Company from Autodesk, which is an adventure for another paper some day.[4]

2.3 The Penn Years

Needing to recover from my recovery, I decided to return to school, and entered the PhD program at the University of Pennsylvania. Inevitably, the EROS project came with me. By this point some questions had emerged in my mind about KeyKOS. The main issue concerned a claim that we had made in a paper entitled *The KeyKOS NanoKernel Architecture* in 1992.[5] Though we claimed that the KeyKOS system was fast, the performance numbers in that paper were (at best) ambiguous. In hindsight, that paper actually did a lot of damage to my faith in the viability of the architecture. There was a lot of variance to account for, and some of it was in places that I knew from my experience with UNIX were important.

But by far the most damaging statement in that paper in retrospect was the statement that KeyNIX (the KeyKOS UNIX personality) ran as fast as the UNIX personality for Mach on the same hardware. At the time, Mach was the standard of reference for microkernel performance, but by the time I got to the University of Pennsylvania I knew a bit more about the Mach system. Claiming that the KeyKOS kernel was fast because it compared favorably to Mach is kind of like claiming that a sports car is fast because it competes favorably with a Trabant. For American readers, I should explain that the Trabant is the East-German knockoff of the Yugo. Definitely not a performance vehicle!

The GNU Hurd team has independently discovered this problem, and is shifting to L4 as their mi

3 Anne Hardy, who had been VP of software at Key Logic, had more patience than I did. Her new company, Agorics, Inc., would finally manage to get a KeyKOS license about 10 years later.

4 See *The Curse of Xanadu*, Wired, Issue 3.06, June 1995. The article is now online. http://www.wired.com/wired/archive/3.06/xanadu_pr.html

5 See http://www.cis.upenn.edu/~KeyKOS/, in particular *The KeyKOS NanoKernel Architecture*.

crokernel to resolve it. And of course, in 1993 Jochen Liedtke would publish the first of his series of papers on the L3/L4 systems. *Improving IPC by Kernel Design* showed that, if anything, Mach was *much* worse than I suspected.

To put my level of concern in perspective, there were basically three historical reasons why capability systems had not taken hold:

1. The belief, validated by several poor implementations, that capability systems didn't perform well in the absence of hardware support.
2. The belief, supported by a key paper written by Earl Boebert, that capability systems couldn't enforce mandatory security policies.
3. The simple fact that no one had really demonstrated one in practice. The notable exception to this was the often-overlooked and very successful AS/400 system, but the OS/400 operating system goes to great lengths to implement a non-capability operating system, successfully obscuring the more powerful mechanisms of the underlying machine.

So I set out to create a publicly available system that would refute all three issues. Today, the EROS system runs on commodity hardware (IA32 family) and is as fast as L4 but enforces protection where L4 does not. In collaboration with Sam Weber, I built a formal verification proof that EROS enforces something called the "Confinement Property." If you can do confinement, mandatory security is relatively easy. In consequence, Earl Boebert has retracted his assertion.

Finally, my students have built a large part of a system that was based "purely" on capability ideas – in the process running into some *real* (but minor) issues that have led us to move on to Coyotos. The code base that Charlie Landau picked up for the CapROS project includes a relatively high performance networking stack, a secure window system, and the beginnings of a base for application construction. I'll talk about what is missing later in this paper.

2.4 The Research Question

The research questions in the EROS project are:

1. Can we build a fast capability system on commodity hardware?
2. Can we preserve performance in places like networking stacks where the system needs to be defensive in a multilayered way?
3. Can capabilities successfully enforce higher-level security policies?
4. Can we design a system that will substantially reduce the propagation of viruses and other threats arising from dynamic content?
5. What would such a system actually look like in practice?

The answer to the first four questions is "yes." The last question is not yet definitively answered, but the shape of the system is certainly starting to emerge.

3 How EROS Is Structured

EROS is structured in a very different way from a more conventional system such as Windows or UNIX. It is somewhat closer to the type of environment that a CORBA or OLE developer deals with, but with the object model extended to deal with security issues and *without* an object request broker.

3.1 An Object-Based System

EROS is an object-based system. The system is structured as a space of objects that are connected by capabilities. Capabilities provide the basic mechanism for object naming and reference, and they also provide the foundational mechanism for protection. Capabilities are protected by the operating system kernel, and cannot be forged.

In some respects, the "feel" of an EROS system is similar to a running Java application where objects are connected and referenced by strongly typed pointers that are protected by a virtual machine. EROS implements transparent persistence – the entire state of the machine is efficiently and consistently saved in the background every five minutes. Like Java, the system is "memory safe." Capabilities are protected by the kernel and cannot be forged. In *contrast* to Java, where the contents of

the runtime image are loaded from files, the EROS runtime image simply continues to run across system shutdown and restart.

EROS is object-based rather than object-oriented. There is no concept of "class." EROS implements interface inheritance, but not implementation inheritance. An EROS capability may be thought of as a protected (facet ID, representation ID) pair, where the facet ID tells the implementing server what operations are authorized by this capability and the representation ID tells the implementing server the name of the representation state of the object. The representation ID only has meaning to the implementing server.

The notion that "everything is an object" is extended to the kernel as well. Services implemented to the kernel are exposed to applications by capabilities. This is carried out all the way down to the level of individual pages, which ensures a uniform interface and protection model for all resources. In consequence, the kernel implements only one system call: "invoke capability." In general, an application cannot tell and does not care whether it is invoking a service provided by the kernel or a service implemented by a user-level object server. EROS is a microkernel-based system; the majority of the system function is implemented by application level code.

Three examples of EROS objects may be helpful to illustrate the different ways in which the facet concept is used in EROS:

Page capabilities name individual pages of disk storage. The kernel is responsible for moving pages in and out of main memory as they are used. Main memory is used purely as a cache of the persistent system state. The "server" for these objects is the kernel, and the kernel implements two facets on each page: read-write and read-only. All kernel capabilities that name memory abstractions include some common methods as part of their facets. One of these is a "downgrade" method – given a read-write capability, the holder can request a read-only capability to the same object.

Space Bank capabilities name storage allocators. A space bank has the authority to allocate pages of storage. All space banks are implemented by a single server; the representation ID mechanism is used in an extended way to record a pointer to the server's internal per-bank state. The space bank server is a trusted server that is part of the system wide trusted computing base.

Directory capabilities name directory instances. In contrast to space banks, the directory implementation is untrusted. Each directory is implemented by a new copy of the directory server. This ensures that two directories cannot communicate with each other about their content unless one holds a capability to the other. We refer to this practice as *polyinstantiation*. It is one of the basic mechanisms for security isolation in an EROS system.

3.2 Persistence

Outside of a few special-purpose embedded systems, a key question in any operating system is "How do things get written down?" More precisely, the question is "How do I get back what I was doing when the system is shut down and restarted?" This is the question of persistence.

3.2.1 The File System Approach

In UNIX, Windows, and most other operating systems, persistence is handled by breaking the system into two layers. The file system layer implements files and directories. These objects are passive content, and they survive system shutdown and restart. Access control lists are recorded to determine which processes can access them. It is entirely the responsibility of applications to decide when to write changes to the disk, and there is no system-level support for guaranteeing consistency of changes to two or more files.

Above this layer is the dynamic layer of the system, which lives only in main memory (possibly extended by a swap area). This layer is the layer where processes, memory maps, and network connections exist. None of this state survives system shutdown.

While simple, there are several problems with this approach:

- In order to save state, an application must write the state to the file system. This has the consequence of "publishing" the state in such a way that it can be examined, modified, or manipulat

ed. Sometimes this type of publication is intend ed, but many times it is mandated by the need to remember rather than any intent to publish.

- There is a loss of security. The need to place state in the file system requires that it be visible. If the state is private, the mere publication of its existence in a way that can be observed by a third party is already a loss of security.
- There is a loss of integrity. In a system using ac cess control lists, there is no way to express the restriction that a file of a certain type should only be manipulated by a certain application. One can only describe the *users* who can access the file – in fact, the application usually obtains its authority from the user rather than the other way around. When integrity is important, we would like to express this the other way around. We would like to say that the *application* has access to the data and the user has access to the application.
- There is a loss of expressiveness. Because ap plications derive their authority from users, there is no way to say that two applications exe cuted by the same user should perhaps have dif ferent authority.

I am not suggesting that there is no need for a file system. It seems obvious that there must be some mechanism that lets users associated human-com prehensible names with objects. However, this is a mechanism of human convenience, and it should not be a requirement that an object be named in a human way in order to be saved.

UNIX programmers may find themselves skeptical about this assertion. As an example of the prob lem, consider the C library's *mktemp*() function which attempts to produce a unique temporary file name. This requirement to name these files has been a source of several security problems in vari ous UNIX implementations, and continues to be awkward today. It would make more sense to have a *mktmpfile*() routine that created an anonymous file and returned a file descriptor that was never visible in any other process? This approach would be intrinsically more secure, and would not rely on race-prone and error-prone manipulations of ac cess control lists to provide security.

3.2.2 The Global Persistence Approach

The alternative is to arrange that *everything* in the system is persistent – including processes. This is the approach that EROS takes. Every five minutes or so, the system makes a copy-on-write snapshot of the complete state of the machine. This snapshot is asynchronously written to the disk in the back ground while the user keeps working. You can kick the plug out of the wall, put it back in, and get all of your windows back complete with all of their contents as they existed when the last system snapshot occurred. Individual applications do not need to do anything to guarantee this, and users are completely unaware that it is happening.

Because EROS processes survive system shut down, no file system is required to provide persis tence. The operating system instead implements a transactional block store with only two kinds of objects at the disk level of abstraction: *pages* (which hold user data) and *nodes* (which hold ca pabilities). Human naming services (directories) are provided by applications that act as directory servers. EROS does not have any notion of a "file system root directory" or a universally shared file system at all. An EROS system is simply a very large space of objects that are connected together by capabilities. Surprisingly, the resulting persis tence implementation is both simpler and faster than a conventional file system.[6]

A simple, obvious, but important consequence of this change is that processes no longer have access to resources based on the identity of their user. In fact, it is the other way around: users have access to resources based on the objects they can invoke and the operations that those objects provide. It is still *possible* to organize and name data in the way that a UNIX system does, but it is no longer *re quired*.

A second consequence is that entities bound in the user name space need not be passive. An EROS directory is simply a set of (string, capability) pairs. The capability can name any object at all, in cluding an object implemented by a server. Ob jects in directories are not limited to just files.

6 See *Design Evolution of the EROS Single-Level Store*, avail able at www.eros-os.org.

3.3 Confinement

When everything is dynamic content, it becomes very important to be able to answer the question "If I create one of these objects, it will run code that I may not trust. Where can my data end up?" The structure of the EROS system guarantees that the only way information can move around is by invoking capabilities, so this question can be rendered more precisely: "If I create one of these objects, will it hold any capabilities that give it write permissions to something I do not know about?"

Usually, we would like the answer to be "no." A client object can then create the new object safely and provide to the object only those additional capabilities that it needs. A subsystem (object) that has no unauthorized write authority is said to be *confined*.

EROS provides a utility object called the "constructor" that is responsible for instantiating new objects. Each type of server in the system has an associated constructor. Directory objects, for example, are created by directory constructors. All constructors run identical, trusted code. Constructors are able to reliably determine whether the objects that it builds are confined. Because it runs trusted code, a constructor can be trusted even when the program that it builds is unsafe.

Constructors are the standard way of instantiating new programs in an EROS system. The vast majority of these programs are confined. The style of program development in EROS is to divide programs into multiple object servers, each of which implements some well-defined function within a confined subsystem.

3.4 Mediation

Dynamic content is dangerous. Consider the number of scripting viruses that have been released this year alone.

Because EROS processes to not obtain authority from their users, it is possible to implement a very powerful security mechanism that we refer to as "mediation." This mechanism goes a long way toward eliminating virus penetration and the consequences of scripting viruses.

For example, an integrated development environment (IDE) would be designed as one object that implements the "shell" of the development environment, and many others that implement editor plug-ins for specific file types. Each running plug-in is confined, and has access only to its own window and content.

Scripting viruses are possible because every application that you run on a UNIX or Windows system has the same access to all of your files that you do. This provides one of the two basic mechanisms that enable viruses propagate (the other is penetration attacks). In EROS, this mechanism does not exist. Return, for a moment, to the example of the IDE above. Note that the plug-ins do *not* have enough authority to open or create files.

3.4.1 Shells vs. Applications

In the lexicon of EROS, we make a very strong distinction between a "shell" and an "application." Applications are not trusted to be well-behaved. We assume that they contain errors, and that the content they manipulate may be compromised. For example, a word processor document may contain a scripting virus.

As a rule, we work very hard to avoid giving applications access to anything that they do not absolutely require. For example, the editor plug-in in our IDE has access to its window, but not to the window system. It can create new sub-windows, but it cannot create new top-level windows.

Shells are distinguished applications that are trusted by their users. A shell acts purely as an agent of some user, and acts with all of that user's authority. The primary job of the shell is to create new application instances and to hand those instances something that we call a "power box."

3.4.2 The Power Box

Obviously, an editor plug-in that cannot create a new file has limited value. We must find some way to enable the plug-in to save files in some name space that is accessible to the user, but at the same time we must *not* permit the plug-in to have access to user files in general. There are two ways to accomplish this:

For batch applications, the solution is to create a new directory that is bound in the user's name space, but give *only* this directory to the application. Because the application does not have access to the user name space in general, this ensures that new files can only be created in this controlled directory.

For interactive applications, the solution is to use a file system mediator. The file system mediator acts purely on behalf of the user, and has access to the user's name space. The mediator facet provided to the application implements a "openForRead()" and "openForSave()" methods, but *these methods return capabilities rather than strings*. When the application invokes the *openForRead*() method, the file system mediator presents the user with the "open file" dialog box. If the user chooses to actually open a file, the file system mediator performs the equivalent to the *open*() call and returns the resulting capability to the application. At no point does the untrusted application have direct access to the user name space. The protocol is similar when *openForSave*() is invoked. Similar mediators exist for other name spaces – notably the network connection mediator. The power box is a collection of these mediators.

Consider this interface from the perspective of the virus author. In order to save a virus to the file system or propagate it into other files, the virus writer must convince the user to agree to write to the file. In order to *transmit* the virus, the virus author must similarly convince the user to open up a network connection to the victim machine, and must then penetrate the victim machine. Alternatively, the virus writer must manage to create a document whose content exploits a vulnerability in the deserialization code of the editor in order to compromise the editor itself.

But a compromise to the editor itself is a short-lived compromise, because the editor will eventually be closed, and can only infect the number of files that the user agrees to write. As a rule, the practice in EROS is polyinstantiation: "one document, one editor."

I will not claim that this will put an end to viruses in the world. Penetration attacks remain a serious concern. However, the places where the application author must be defensive are well localized in this type of design, flood-style propagation of viruses becomes impractical, and the achievable damage starting from any given document is greatly reduced – primarily because the *editor* doesn't have enough authority to do any interesting degree of damage.

3.5 Principles

There is a short list of design principles that make EROS effective at defending itself.

Local Names The majority of name spaces that are visible at application level in EROS are local. For example, there is no shared file system. Sharing must be established explicitly.

Uniform Protection All resources are protected using a common mechanism: capabilities. In contrast to access control lists, it is possible to reason formally about authority in a capability system, and to enforce security policies. It has been shown formally that the UNIX security deficiencies cannot be fixed without fundamental and incompatible changes to the UNIX protection model.[7]

Polyinstantiation The best way to avoid leakage of information is never to share state in the first place. The majority of object servers in EROS are "one object, one server."

Mediation Whenever objects are aggregated into collections, applications should not be given unmediated access to the collection.

Limited Disclosure The EROS design is *extremely* privacy-oriented. Where the UNIX design philosophy is "open is good," the EROS design philosophy is "every disclosure of information is a potential mistake and must be viewed skeptically and argued carefully."

This is in keeping with the three fundamental rules of capability-based system design:

- Connectivity begets connectivity. Capabilities are initially conveyed to a process at creation

7 Harrison, Ruzzo, and Ullman, *Protection in Operating Systems*, Communications of the ACM, Vol 19, No 8, August 1976. The really important part of this paper is poorly understood – it isn't the undecidability result. The important part is that all *finite* systems are decidable, all *real* systems are finite, and with every system examined in that paper *except* the capability model, *the outcome is unsafe*.

time, or later by transfer from some other process that already holds a given capability.

- *Only* connectivity begets connectivity. If you don't have a capability to an object, there is no way to manipulate it. In fact, there is no way to determine whether that object exists in the system.
- Capabilities are the *exclusive* means of naming objects. This ensures that connectivity-based analysis and filtering techniques are sufficient to establish and enforce security properties.

3.6 Reduced Role of the Administrator

One consequence of the move to capabilities is that the authority of administrators is greatly reduced. An EROS administrator simply has no way to list the processes running on the system generally. If a given user wishes to disclose their list of activities in order to obtain help it is possible to do so. This is not merely due to the absence of a necessary utility process. The standard system install does not provide the administrator with the necessary authority to *implement* such a utility.

This will seem odd to most UNIX users. Remember that the assumption in the EROS design is that the system is being leased as a computational utility. The role of the administrator is, at most, to provide the physical and computational system resources. It is simply none of the administrator's business what is done with them, and under Common Carrier statutes there are liability-related reasons why an administrator might not *want* to be able to examine what users are doing.

This change is not entirely motivated by security. It also turns out that administrator action is the predominant source of system crashes. The EROS approach tries very hard to be "operator-less."

4 The EROS Nucleus

EROS is now a pure microkernel, but it has gone through several evolutions to get there. In this section I will describe the evolution of the design over time, and the current version of the kernel-level system architecture.

4.1 A Microkernel Design

It has become fashionable since the arrival of Mach to argue whether microkernels or monolithic kernels are a better design approach. In particular, a lot of attention seems to go to whether drivers belong in the kernel or not. This is very distracting, because the important issue in microkernel design isn't size – it is architecture. The important criteria for evaluating a microkernel are the ability to effectively isolate device management, the approach to exception handling policy, whether there is a small, low-level kernel abstraction set, and the choice of execution model (atomic or otherwise).

4.1.1 Drivers

On the IBM System/360, where the GNOSIS history started, the question of in-kernel drivers isn't really very meaningful. Because of the structure of the S/360 I/O system, a complete operating system requires only three or four kernel-level drivers to be completely functional, so the question of whether drivers reside in the kernel or at user level isn't very interesting. Also, the S/360 provided virtual DMA, which significantly reduces the risk of driver-level programming errors. Recent PC evolution has rediscovered virtual DMA under the name "I/O MMU' in the most recent PCI-based systems.

From a kernel design perspective, a significant advantage of S/360 lies in the uniformity of the channel architecture. The S/360 and its successors should be thought of as a heterogeneous multiprocessor – each I/O card is responsible for implementing a common bus-level protocol and a common "channel program" interpreter for its device class. Like the later SCSI design (which borrowed heavily from the S/360 channel architecture), channel programs are "generic." They can be constructed by user-mode applications, validated for safety by a simple kernel-level driver, and shipped to the device for processing. Because of this, GNOSIS needed only three kernel-level drivers: a block device driver, a console device driver, and the "generic channel program" device driver. These were sufficient to boot the kernel, and to build non-privileged drivers for other devices as application code.

Having the block device drivers in the kernel made it possible for GNOSIS to implement checkpoint ing directly in the kernel, which was an important enabler to the overall system architecture. An effi cient checkpoint system requires kernel support. While system wide checkpointing *can* be done in a pure microkernel design, it is much simpler to see how to do it after you have seen the simpler "all in one" design.

As KeyKOS was later ported to workstations and systems with less regular I/O architectures, a larg er number of drivers moved into the kernel. The early ports were to SCSI-based workstations, which shared the convenience of the channel ar chitecture through the SCSI generics mechanism, having only a small number of "special case" de vices such as network interfaces and the display subsystem that were attached directly to the moth erboard.

When the EROS effort began, this approach be came unmanageable. The PC, even in its 2005 form, remains an ugly kludge compared to the S/360 architecture of 1964. At the time we started the EROS project in late 1991, there were thou sands of devices, each having its own ad-hoc pro tocol for determining such basic things as interrupt selection. For developers looking today at the PCI bus and a few surviving legacy devices on the motherboard, it is hard to imagine just how badly designed the original ISA bus was.

While the EISA bus, introduced in 1988, was a significant improvement, the proliferation of 16-bit ISA cards would remain a problem well into the late 1990's, and wouldn't really disappear until the proliferation of the PCI bus. By 1995, it was clear that we needed to push device drivers entirely out side the EROS kernel, but there were interactions between this logic and the checkpoint system that needed to be resolved. By 2002, an "outside the kernel" driver had partially emerged. and the cur rent EROS system implements all of its drivers outside the kernel.

4.1.2 Exception Handling

GNOSIS took a microkernel approach to excep tion handling issues from the beginning. Each pro cess designates (via a capability) another process that serves as its exception handler (in EROS terms, a "keeper"). If the application takes an ex ception, the kernel converts this into a message that is delivered to the keeper.

One difference between the EROS family of ker nels and other microkernels is that memory excep tions are treated differently from execution excep tions. In addition to the fault handler associated with a process, EROS provides a way to associate a fault handler with a particular subregion of mem ory. If an application takes an *invalid access* or *protection violation* exception, the exception is first reported to the responsible memory fault han dler (if present). That handler can either resolve the problem or pass it to the per-process fault han dler.

Because memory faults are distinguished, it is pos sible to factor memory management policy from execution failure policy. For example, an object database can share an object cache with an appli cation, and provide a fault handling policy for that cache that is specific to the needs of the object database implementation.

4.1.3 Kernel Complexity

The C version of the KeyKOS kernel was 20,000 lines of very carefully written C code. By 1996, the EROS kernel consisted of 42,000 lines of C++ code – most of the growth being due to the ver bosity of C++ and the intrinsic complexity of the IA32 and PC architectures. In comparative terms, EROS is significantly simpler than the Mach ker nel, but more complex than the L4 kernel. The re ally significant differences between the L4 imple mentation and the EROS implementation come from the support for persistence and the differ ences in the address space models.

As is usual for a microkernel, the EROS kernel implements a very small set of kernel abstractions consisting of processes, address spaces, schedules, and primitive storage units (pages and nodes). These are discussed in greater detail below.

In contrast to L4, the EROS architecture imple ments a software-defined memory translation hier archy. The kernel is responsible for translating this structure into terms that the hardware can directly manipulate, and ensuring consistency between the

software defined and hardware defined data structures.[8] This is done because we wish to use capabilities as a universal basis for protection, and also because it allows us to represent memory mapping data structures in a way that can safely and efficiently be saved to the disk during a checkpoint operation.

With the exception of the memory management structure, the remainder of the EROS kernel is surprisingly similar in style to the L4 kernel. There are only two other significant differences at this level of the system:

1. EROS uses capabilities rather than thread identifiers to name IPC destinations. This is simply an added level of indirection.

2. EROS implements only one system call, "invoke capability". Instead of invoking kernel services through multiple system calls, services are presented as "objects" that are named by kernel-implemented capabilities.

Naming kernel services with capabilities regularizes the system call path at some cost in performance, but kernel capabilities are very low-level abstractions that are rarely invoked. We expect to improve this performance in the Coyotos system.

4.1.4 Execution Model

One place where EROS differs significantly from other microkernels is its use of an atomic system call design. An EROS system call proceeds in two phases. In the *setup* phase, all preconditions required for completion of a system call must be satisfied and all necessary permissions checks must be performed. Internal caches and data structures can be updated, but no externally visible "logical" modification of system state is permitted during this phase. If an unavailable resource must be obtained, the operation necessary to obtain the resource is started and the invoking process is put to sleep on the appropriate stall queue. Sleeping processes do not retain any stack in the kernel – when awakened, the system call will be restarted from the beginning. The EROS kernel has one kernel stack per CPU, not one kernel stack per process.

On every path through the kernel there is an explicitly identified "commit point." This marks the end of the setup phase. After the commit point is the *action* phase. During the action phase, logical modifications to the system state are permitted, but the operation cannot block or demand new resources. If an operation cannot, for some reason, be completed during the action phase, *the kernel must halt entirely.* Completion here might mean "return an error code," but the idea is that every update to the overall system state should be an "all or nothing" transformation.

Advantages In an atomic system call design, system calls are short and must involve few resources. However, there is no possibility of kernel deadlock, and system-wide consistency is relatively straightforward to verify.

For example, the atomic system call structure allowed us to use model checking techniques to verify that several critical invariants were actually satisfied by the EROS implementation.[9] This is possible in part because the atomic design approach eliminates the alias analysis challenges that would otherwise make this type of check impractical or even impossible.

Because of their simplicity and precise resource management discipline, atomic designs are fairly simple to extend to multiprocessor implementations than conventional designs.

Disadvantages When properly implemented, there are few disadvantages to a purely atomic system call design, but there are a few unusual constraints that are imposed on the kernel implementation:

1. Every operation must be prepared to abandon its work at any moment prior to the commit point. In particular, this means that arrangements must be made to return any dynamically allocated storage if the system call is abandoned, and to ensure that in this event there are

8 Jonathan .S. Shapiro, David J. Farber, and Jonathan M. Smith, "State Caching in the EROS Kernel," *Proc 7th International Workshop on Persistent Object Systems*, Cape May, NJ 1996.

9 Hao Chen and Jonathan S. Shapiro, "Using Build-Integrated Static Checking to Preserve Correctness Invariants," *Proc 11th ACM Conference on Computer and Communications Security*, Washington, D.C., 2004

no dangling pointers. Locks must also be released.

2. Linux-style drivers cannot easily be implemented within the kernel, because they rely on the ability to sleep within the kernel with a retained kernel stack. There are several possible ways to resolve this. The most straightforward solution is to run these drivers as application code.

In the EROS implementation, restarting a system call is accomplished by branching directly to a small piece of assembly code that simply resets the kernel stack pointer for the current per-CPU stack to the top of the stack (i.e. the stack is now empty). A helper routine is then called to deschedule the current thread (the one that just yielded) and clean up any temporary state associated with the current kernel invocation – this amounts to only three or four variables. Of the four, three are cleared for paranoia reasons.

Performance Ford *et al.* have published a paper comparing the performance of "interrupt model" and "process model" kernels.[10] They conclude from their measurements that neither model offers a significant performance advantage over the other. This conclusion should be viewed with skepticism, because the performance measurements revealed in their conference talk show that the continuation restart time of their implementation exceeds the total end-to-end EROS path length by a significant multiple.

Microbenchmark measurements of EROS performance show that it is just as fast as Linux. One reason that EROS's atomic system call performance is faster than the interrupt model examined by Ford *et al.*, is that the approach is different. Ford's interrupt model seeks to preserve partial progress that has been made by a system call. This is done by dividing the call into a sequence of atomic steps and explicitly storing any information needed by the next stage as part of the per-process kernel state. The EROS approach abandons this state altogether, preferring to succeed or fail on the entire system call.

4.2 Kernel-Implemented Services

The main services implemented by the EROS kernel are processes, address spaces and scheduling. The kernel is also responsible for handling interrupts and exceptions by converting them into messages that are delivered to some handler process. In short, the set of kernel abstractions is the standard set implemented by most microkernels.

There are four significant differences between the EROS nucleus and other microkernel implementations: capabilities, the address space architecture, the persistence mechanism, and an unusual set of access rights.

4.2.1 Capabilities and Invocation

Most kernels (and most microkernels) implement several system calls for each object. When an application invokes the kernel, the pattern of execution is first to determine what operation is being performed (what system call) and then to decode the arguments of the call to determine what object is being acted on. Once the object is known, a permissions check is performed to determine if the invoking process has the necessary permission to perform the requested operation on that object.

In a capability system, the order of decoding is reversed. The capability being invoked is decoded first, which provides the identity of the object and the permissions that apply to this operation. Control is then dispatched to an object-specific decoder that determines the operation to be performed.

EROS uses capabilities to name *everything*. All system resources down to the individual page of data are named by capabilities. This arrangement is true both in memory and on the disk. Even CPU schedules are named by capabilities. The scheduling class of a process is discovered by examining a dedicated "slot" in the process state that contains the scheduling capability.

In EROS, there is only one system call: invoke capability. This system call has three variations, CALL, RETURN, and SEND, that are intended to support

10 Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullman, "Interface and Execution Models in the Fluke Kernel," *Proc. 3rd Symposium on Operating System Design and Implementation*, New Orleans, LA, 1999.

interactions in the style of interprocess procedure calls. The CALL operation invokes a capability and waits for a response ("send and wait"). The RETURN operation invokes a capability and waits for the next incoming request ("reply and receive"). The SEND operation transmits a message but does not expect a response – it serves to initiate a new, independent thread of control. As with L4 and other third generation microkernel designs, capability invocations are synchronous, blocking and unbuffered.

The invoke capability call has a regularized message payload. The sender transmits four capabilities, four data registers, and a contiguous string of up to 65 kilobytes. The interpretation of the message content is up to the receiver, but by strong convention data register zero is used for the *request code* or the *result code*. In similar fashion, capability register zero is used to provide a reply capability. Because of this convention, it is possible to build generic "front end" objects that can forward messages without knowing their contents. This can be very useful for performing live object upgrades.

4.2.2 Address Space Model

In contrast to most microkernels, the EROS implements a software-defined address space architecture. An EROS address space is a hierarchical mapping structure that is similar to the ones used in many current hardware architectures. Page tables are replaced in the software mechanism by *nodes.* Just as an IA32 page table holds 1024 page table entries, a node holds 32 capabilities. Applications construct address spaces by acquiring nodes from their space banks and assembling them into an appropriate hierarchical arrangement. This mechanism provides fine grain mapping control to the application, including full support for page table and page directory sharing.

Rationale There are two reasons that this approach is used in EROS.

First, it allows mapping structures to be specified using capability-defined structures. This preserves the general use of capabilities throughout the system, and ensures that address spaces can be saved in the same way that most other system data structures are saved.

Second, it ensures that the kernel is not responsible for allocating the storage that is used to describe mapping tables – all such data structures are explicitly allocated from a user-level storage allocator. This is in contrast to the L4 *map* operation or the UNIX *mmap* operation, both of which require the kernel to allocate book-keeping storage, the EROS address space mechanism does not require any kernel storage allocation for mappings to be constructed, and ensures that all storage is properly accounted for.

More precisely, the kernel maintains a fixed pool of page tables that are allocated at system initialization time. *These tables are not definitive*. They can be discarded at any time, because their content can always be reconstructed from the node structures. As with the kernel process table, the hardware page table structures are a cache – the definitive statement of the mapping structure is the one encoded in the software defined address space structures.

This brings to light an important design restriction in the EROS system: the kernel does not allocate storage on behalf of user operations. The kernel may load and unload software cache entries in order to support the actions of applications, but the content of these caches *always* originates in some data structure that has been allocated at user level and has been paid for by the application. *No denial of resource attacks against the kernel are possible*. We are often asked to cite the Stanford Cache Kernel as the inventor of this idea. Examination of the citations in their papers shows that they got the idea from KeyKOS.

Fault Handling One useful advantage of a software-defined address space representation is that an application controlling a portion of an address space can define a fault handler for that sub-space. This ability can be very useful in a relationship where one party must supply the storage while a second controls the fault handling policy.

4.2.3 Persistence

The EROS kernel provides the basis for persistence via a kernel-implemented snapshot mecha

nism. At periodic intervals, a trusted user-level application declares that a checkpoint should be taken by invoking a kernel capability. The kernel immediately marks all modified objects "copy on write," and cooperates with user-level drivers to asynchronously write these objects to the disk.

The current EROS implementation actually traverses all of the objects in memory to implement this, which requires time proportional to the amount of memory. A better, simpler, but currently unimplemented design would handle this operation incrementally, allowing a snapshot to be declared in unit time and distributing the burden of workload in a way that is proportional to usage.

4.2.4 Access Rights

The basic capability access restrictions in EROS are READ-ONLY, NO-CALL, and WEAK. The read-only restriction ensures that the object named by the capability cannot be modified. The NO-CALL restriction is used in address spaces to indicate that subspace fault handlers are untrusted and should not be invoked. The WEAK access restriction is unique to EROS and Coyotos, and addresses a problem specific to capability systems.

A capability system consists of an object graph. One difficulty with enforcing access control in an object graph is that a READ-ONLY restriction isn't enough. Suppose that two processes A and B are not supposed to be able to communicate. Both hold READ-ONLY hold capabilities to some node N. If the node N in turn contains a capability that allows writes to some object, A and B can communicate indirectly through this second object. A similar problem exists in the C programming language: a constant structure can contain a non-constant pointer. In an efficient capability system, some mechanism is needed to enforce transitive write restrictions.

The WEAK access restriction has the effect of "downgrading" every capability that is fetched. If process A holds a WEAK capability to a node N, and fetches a capability from that node, the capability actually returned is guaranteed to have the READ-ONLY and WEAK restrictions set. This transformation is performed by the capability copy operation. The effect of this constraint is that a WEAK capability is *transitively* read-only. This is necessary, for example, to efficiently support copy-on-write sharing of address spaces between two processes that are not supposed to communicate.

5 Towards A Secure Browser

To validate the EROS architecture, we have implemented a small number of early applications. In particular, we crafted a browser application that fetches content from the network over a secure protocol stack and displays it through our secure window system using a fully isolated application (in the sense of section 3.4.1).

5.1 The Window System

The browser communicates with the user using the EROS Window System (EWS).[11] This window system is part of the trusted computing base for all applications. The entire window system is implemented in about 4,500 lines of trusted code. Most responsibility for rendering is handled in the untrusted client.

One feature supported directly by EWS is the notion of sessions and sub-sessions. A graphical shell can create a window for use by a subordinate, untrusted plug-in. This window acts as the "root window" for the plug-in. A second key feature is that all of the storage used for rendering and visualization purposes is allocated from a client-supplied space bank.

The protocol between EWS and its clients is very simple. The client supplies storage for a shared memory region. Using the supplied space bank, the window manager invokes a region constructor to fabricate a memory subspace that will be shared with the client. This protocol ensures that the window manager controls the fault handling policy for the region. The window manager supplies the resulting region back to the client, and the client renders into the region.

When rendering is needed, the window manager performs `bitblt` operations from the region into the hardware frame buffer. The client is not given any notification of expose events. When the client

11 Jonathan S. Shapiro, John Vanderburgh, and Eric Northup. "The Design of the EROS Trusted Window System," *Proc. 2004 USENIX Security Conference*.

has updated a region and wishes the update displayed, it directs the window system to redraw all of the area within a given bounding box. This reverses the relationship present in the X11 server, and eliminates a very bad communication channel that is inherent in the X11 design. The design can be extended for high-performance 3D rendering.

The `bitblt` algorithm is the *only* rendering algorithm implemented by the window manager. It is unlikely that there are exploitable flaws in the algorithm itself, and EWS is prepared to recover gracefully if a hostile client unmaps the region.

5.2 The Network Stack

The network stack[12] used in the prototype provides a demonstration of EROS's ability to provide "defense in depth." There are several layers of protection between the ethernet wire and the application. Of these, only the ethernet driver has any potential to globally compromise the system (because it has access to physical DMA). The TCP/IP stack is polyinstantiated. Compromising a given network stack will only compromise a single application.

The most interesting point about this network stack is that its performance is very nearly as good as the native Linux networking stack. On gigabit networks it sacrifices about 10% of peak throughput for greatly improved protection. Addressing this performance disadvantage requires architectural changes to the EROS interprocess communication mechanism that will be implemented in the EROS successor.

5.3 The Browser

The browser itself is divided into a shell and a rendering plug-in. The current implementation of the browser renders plain ASCII text, primarily because we ran out of time in the prototype because of portability difficulties that are discussed below. However, there is no reason to anticipate any deep technical difficulty in porting the KHTML implementation or a similar browser to run in this environment.

Because the browser plug-in has essentially no authority to examine or modify other state on the user's machine, we are not greatly concerned about scripting viruses. A hostile script can alter what is rendered to the user, but only within its own window. It cannot modify local files or open network connections without user consent.

Ultimately, these same restrictions would apply to native code. This means that it is feasible within the framework of the EROS browser to implement Active-X style plug-ins downloaded from remote locations that cannot compromise the local system.

6 Other Security Approaches

It seems worthwhile to ask whether other methods might accomplish the types of isolation that EROS achieves – perhaps without a completely incompatible API. It is clear that access control lists alone are insufficient, but the question of what might be done using Linux Security Modules (LSM) seems worth exploring.

In particular, what might happen if we used LSM to prevent most programs from opening files or sockets directly, but we permitted the transfer of descriptors from shells to applications? We might then use a shell architecture similar to the one outlined above, where an open/save-as agent performs the open and passes a descriptor back to the application. If this design is carried through to its logical extreme, we would be left with principal-based authentication as a largely vestigial artifact of the early UNIX design. It seems likely that an augmented interprocess communication mechanism would become necessary so that descriptors might more easily be transferred – for example by extending the local stream abstraction.

It is possible that this approach would work, but I am skeptical. The LSM design is intended to solve static rather than dynamic security configuration problems. The reason that a hybrid design might be feasible is that the list of shells is a mostly static list of applications, and that all other applications could simply be treated as "confined." I have briefly discussed this possibility with Stephen Smalley (the author of the SE-Linux policy), and he is also skeptical. Neither of us believes that the *idea* is unsound, but both of us share the intuition

12 Anshumal Sinha, Sandeep Sarat, and Jonathan S. Shapiro, "Network Subsystems Reloaded: A High-Performance, Defensible Network Subsystem." *Proc. 2004 USENIX Annual Technical Conference.*

that LSM is better suited for static policy specifi cations. It seems worth exploring this further.

In practice, the major impediment would be an is sue of programming idiom. Current UNIX appli cations assume that they have very liberal access to the file system. Changing this would require significant application restructuring that is likely to be non-portable.

Finally, it should be pointed out that this approach would provide the *security* benefits of isolation, but not the *integrity* benefits of the EROS ap proach. The EROS integrity advantage rests in the fact that applications can hold descriptors that are not obtained from their users, and that these de scriptors persist across system restarts. This cannot be practically achieved in UNIX. The UNIX archi tecture wasn't designed to support it, and lacks the necessary provisions for data structure integrity checks that are necessary to sustain a long-term checkpointing design.

7 Weaknesses in EROS

The EROS architecture has met nearly all of the objectives that we set for it technically. There are a number of weaknesses in the current kernel archi tecture:

- Multithreaded applications are not supported well. A better design would use first-class com munication endpoints in place of sending mes sages to processes directly. Such a design would also simplify the persistence implementation.
- Asynchronously communicating processes are not supported efficiently. It is easy enough to set up a shared memory region, but there is no efficient mechanism by which a sender can noti fy a receiver that the shared buffer contains con tent to be processed. This is the main source of performance loss in the current defensible net working stack.
- The software address space data structure is less flexible than it might be, and requires more overhead structures than are really required for both large and small processes.

The solutions to each of these issues is straightfor ward. The introdution of endpoints will require re thinking capability invocation in certain ways, but existing applications will not be greatly changed. All three of these issues are being addressed in the Coyotos system.

The major impediment to bringing up applications on EROS is the simple fact that EROS is not UNIX. To our surprise, the real difficulty is the widespread use of tools like `autoconf` and `libtool`. Autoconf in particular is *very* widely used. It is extremely UNIX dependent, and it does not support cross compilation environments ade quately. In many cases there are libraries we would like to port where the library itself presents minimal difficulty, but the autoconfiguration pro cess cannot be made to run.

Our solution to this in Coyotos will be to finally implement a UNIX-like build environment so that we can port software more easily. While EROS was an active research project, we avoided this de liberately. Our observation was that once the UNIX personality was constructed for Mach, very little research happened using the Mach platform directly. Since our research goal was to evaluate the EROS platform, we were concerned that bring ing up a POSIX environment would be self defeat ing.

From a security perspective, the overwhelming weakness in EROS its that the security-critical code is implemented in C. The problem with this is that C code cannot be formally analyzed, so it is impossible to know rigorously whether this code is correct. We do have strong anecdotal and model-checking evidence that the code is quite good, but this is not the same thing as being able to *prove* it. Fixing this issue is a major objective for the Coy otos project.

Many people have asked us what we plan to do for applications if we aren't running a UNIX-compati ble API. This question fails to notice that nearly all interactive applications today are written to a *toolkit* API rather than an operating system API. The toolkit may be GTK or QT, or it may be something else. So the right question is: can one of these toolkits be ported to an EROS-like system with minimal damage?

As it happens, *all* of the major toolkit APIs currently in use are designed around the use of CORBA-based objects. This style of system structure is very friendly to the type of implementation that EROS encourages. It seems likely that by encouraging some very minor, portable changes in the current toolkit APIs we may be able to move applications very easily. Since these changes would improve security on conventional systems, and do not involve major invasive changes, it is possible that the toolkit builders will see them as intrinsically worthwhile.

8 Conclusions

EROS is a very different system from most of the systems that readers may already know about. It provides a degree of security and integrity control that is not achievable in conventional operating systems such as Linux, and it is extremely robust.

The current EROS implementation is well-suited to embedded deployments. With minor work to restore function that was temporarily removed, it could provide the basis of a desktop or server system. The CapROS project is trying to do this, and I very much look forward to seeing what they will create.

In the laboratory that produced EROS, work has turned to the Coyotos operating system and the BitC programming language, which are the topic of another talk at this conference.

9 Acknowledgments

No discussion of EROS would be complete without acknowledging the many contributions of the KeyKOS architects: Norm Hardy, Charlie Landau, and Bill Frantz. All three have been very active participants on the EROS design lists, and all have taken the time to explain various aspects of the KeyKOS design. There are also many people on the EROS-related mailing lists who have helped along the way.

John Vanderburgh implemented the EROS Window System. Anshumal Sinha implemented the defensible network stack in collaboration with Sandeep Sarat. Eric Northup and Swaroop Sridhar contributed significantly to both efforts.