EROS: A Principle-Driven Operating System from the Ground Up

Jonathan S. Shapiro, *Johns Hopkins University*

Norm Hardy, *Agorics, Inc.*

Design principles are highly advocated in software construction but are rarely systematically applied. The authors describe the principles on which they built an operating system from the ground up, and how those principles affected the design, application structure, and system security and testability.

The Extremely Reliable Operating System[1] is a capability-based operating system designed to support the security and reliability needs of active systems. Users of active systems can introduce and run arbitrary code at any time, including code that is broken or even hostile. Active systems are shared platforms, so they must simultaneously support potentially adversarial users on a single machine at the same time.

Because active systems run user-supplied code, we cannot rely on boundary security to keep out hostile code. In the face of such code, EROS provides both security and performance guarantees (see www.eros-os.org for downloadable software). An application that executes hostile code (such as viruses) cannot harm other users or the system as a whole and cannot exploit a user's authority so as to compromise other parts of the user's environment.

The EROS project started as a cleanroom reimplementation of KeyKOS,[2] an operating system Norm Hardy and his colleagues created for the IBM System/370 (see www.cis.upenn.edu/~KeyKOS for earlier documents from the KeyKOS system). The key contributions of EROS are formal verification of some of the architecture's critical security properties and performance engineering. These security and performance capabilities come from two sources.

First, the primary system architecture is uncompromisingly principle-driven. Wherever a desired feature collided with a security principle, we consistently rejected the feature. The result is a small, internally consistent architecture whose behavior is well specified and lends itself to a careful and robust implementation. Second, the system's lead architects had prior experience as processor architects. This helped us avoid certain kinds of abstraction that modern operating systems generally include and seek a design that maps directly onto the features that modern hardware implementations provide; very little performance is lost in translating abstractions.

Figure 1 shows the core EROS design principles. Jeremy Saltzer and Michael Schroeder first enumerated many of these in connection with the Multics project[3] and incorporated others based on our experiences from other projects (see the "Related Work" sidebar). There are no magic bullets in the principles we adopted for EROS. The system's perform-

0740-7459/02/$17.00 © 2002 IEEE

ance and design coherency results solely from finding better ways to adhere consistently to these principles at fine granularity without sacrificing performance.

We maintained strong adherence to design principles in the EROS/KeyKOS design for three reasons:

- We wanted to know that the system worked and why it worked. Unless you can trace each piece of the system code back to a motivating principle or a necessary correctness constraint, achieving this is difficult. Traceability of this type is also required for high-assurance evaluation.
- We expected that a clean design would lead to a high-performance implementation. Based on microbenchmarks, this expectation has been validated.[1]
- We wanted to formally and rigorously verify some of the security mechanisms on which the system relies. A rigorous verification of the EROS confinement mechanism, which is a critical security component in the system, was recently completed.[4]

This article provides some examples of how these principles affected the EROS system design. We also describe the application structure that naturally emerged in the resulting system and how this affected the system's security and testability.

The EROS kernel design

The most direct impact of design principles in EROS is in the kernel's structure and implementation. In several cases, our strict adherence to design principles led to unusual design outcomes, some of which we discuss here. (Except where made clear by context, references to the EROS system throughout the article refer interchangeably to both EROS and KeyKOS.)

Principles from the Multics Project

- *Economy of mechanism*: Keep the design as simple as possible.
- *Fail-safe defaults*: Base access decisions on permission rather than exclusion.
- *Complete mediation*: Check every access for authority.
- *Open design*: The design should not be secret. (In EROS, both design and implementation are public.)
- *Least privilege*: Components should have no more authority than they require (and sometimes less).
- *Least common mechanism*: Minimize the amount of shared instances in the system.

Commonly accepted principles

- *Separation of policy and mechanism*: The kernel should implement the mechanism by which resource controls are enforced but should not define the policy under which those controls are exercised.
- *Least astonishment*: The system's behavior should match what is naively expected.
- *Complete accountability*: All real resources held by an application must come from some accounted pool.
- *Safe restart*: On restart, the system must either already have, or be able to rapidly establish, a consistent and secure execution state.
- *Reproducibility*: Correct operations should produce identical results regardless of workload.

Principles specific to EROS

- *Credible policy*: If a security policy cannot be implemented by correct application of the system's protection mechanisms, do not claim to enforce it.
- *No kernel allocation*: The kernel is explicitly prohibited from creating or destroying resources. It is free, however, to use main memory as a dynamic cache for these resources.
- *Atomicity of operations*: All operations the kernel performs are atomic—either they execute to completion in bounded time, or they have no observable effect.
- *Relinquishable authority*: If an application holds some authority, it should (in principle) be able to voluntarily reduce this authority.
- *Stateless kernel*: The system's security and execution state should logically reside in user-allocated storage. The kernel is free to cache this state.
- *Explicit authority designation*: Every operation that uses authority should explicitly designate the source of the authority it is using.

Figure 1. Core EROS design principles.

Safe restart

In secure systems, we must ensure that the system has restarted in a consistent and secure state. In most operating systems, there is an initial set of processes that the kernel specially creates. These processes perform consistency checks, reduce their authorities to their intended steady-state authority, and then initiate the rest of the programs in the system. This creates two problems:

1. The consistency checks are heuristic, which makes establishing their correctness difficult. The Unix `fsck` command, for example, must decide which files to throw away and which to keep without knowing how these files interrelate. Consequently, the state of the group and password files might not be consistent with each other.

2. The initial processes receive their authority by means that are outside the normal mechanisms of granting or transferring authority. The designers must make specialized arguments to demonstrate that the system appropriately manages and diminishes this authority. The complexity of these arguments is comparable to the complexity of the correctness arguments for the remainder of the system.

EROS resolves both issues by using a transacted checkpointing system. The system periodically takes an efficient, asynchronous snapshot of the entire state of the machine, performs a consistency check on this state, and then writes it down as a single disk transaction. Because the system is transacted as a whole, no possibility of global inconsistency exists. On restart, the system simply reloads the last completed transaction. System installation consists of writing (by hand) an initial system image; the processes of this system image have no unusual authority.

Stateless kernel

EROS is a stateless kernel—the system's execution state resides in user-allocated storage. The kernel achieves performance by caching this state. A caching design facilitates checkpointing and imposes a dependency tracking discipline on the kernel. To ensure that user-allocated storage always reveals correct values when examined, the kernel must be able to restore this state on demand. These dependencies provide a form of self-checking. The kernel can sometimes compare its cached state to the user state to detect whether the runtime kernel state has become inconsistent, preventing a bad state from transacting to disk.

EROS does not publish a memory map abstraction, because this would violate the stateless kernel principle. Instead, EROS requires that the applications explicitly allocate all of the pieces that comprise the mapping structure. Figure 2 shows a small EROS address space. The application explicitly allocates (typically by a user-level fault handler) every node and page in this address space. The kernel builds the hardware-memory-mapping tables by traversing this structure and caching the results in the hardware-mapping tables.

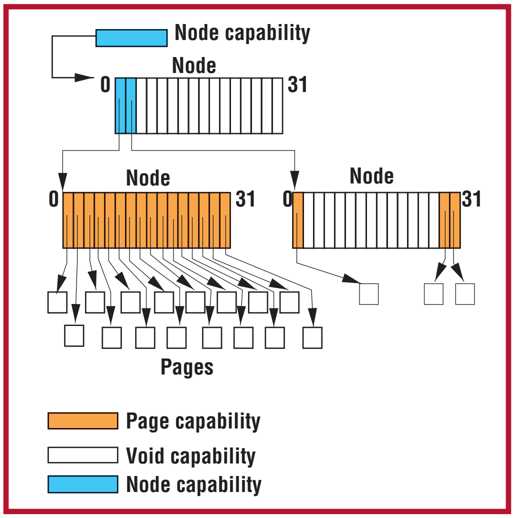


Figure 2. An EROS address space.

Related Work

Henry Levy[1] and Ed Gehringer[2] provide overviews of several capability systems. EROS borrows ideas directly from three prior capability systems. Like Hydra,[3] EROS is an extensible capability system. Programs can implement new objects that protected capabilities invoke. Like CAL/TSS,[4] EROS unifies processes with protection domains. EROS designers also took to heart most of the design lessons reported from the CAL/TSS project. The Cambridge CAP computer,[5] while implemented in hardware, similarly used fine-grain capabilities for memory protection. It's also the first example of a stateless kernel.

EROS uses kernel-protected capabilities. An alternative Amoeba[6] uses treats capabilities as data, using unguessably sparse allocation for protection. This approach does not support confinement, because it is impossible to determine which bits of the application represent data and which represent capabilities.

Simple cryptographic or signature schemes share this problem. One solution is password capabilities as used in Monash[7] and Mungi,[8] which apply a system-defined XOR before accepting capabilities. A concern with this approach is that any operation simple enough to be efficient (such as XOR) is easily reverse-engineered. True cryptographic checks must be cached to avoid prohibitive computational cost.

References

1. H.M. Levy, *Capability-Based Computer Systems*, Digital Press, 1984.
2. E.F. Gehringer, *Capability Architectures and Small Objects*, UMI Research Press, Ann Arbor, Mich., 1982.
3. W.A. Wulf, R. Levin, and S.P. Harbison, *HYDRA/C.mmp: An Experimental Computer System*, McGraw Hill, New York, 1981.
4. B.W. Lampson and H.E. Sturgis, "Reflections on an Operating System Design," *Comm. ACM*, vol. 19, no. 4, May 1976, pp. 251–265.
5. M.V. Wilkes and R.M. Needham, *The Cambridge CAP Computer and its Operating System*, Elsevier, North Holland, 1979.
6. A.S. Tannenbaum, S.J. Mullender, and R. van Renesse, "Using Sparse Capabilities in a Distributed Operating System," *Proc. 9th Int'l Symp. Distributed Computing Systems*, IEEE Press, Piscataway, N.J., 1986, pp. 558–563.
7. M. Anderson, R. Pose, and C.S. Wallace, "A Password Capability System," *The Computer J.*, vol. 29, no. 1, 1986, pp. 1–8.
8. G. Heiser et al., "Mungi: A Distributed Single Address-Space Operating System," *Proc. 17th Australiaision Computer Science Conf.*, ACM Press, New York, 1994, pp. 271–280.

Table 1

Protection properties of capabilities

	Conventional systems	Capability systems
Based on	(`User`, `object`) pair	Per-process capabilities
Lookup by	`rights(Object, process.user)`	`rights(process[cap ndx])`
Authority grant	Program run by owning user can grant authority to anyone	Can transfer if an authorized path of communication exists
Name resolution	String lookup (via `open`)	Direct designation

Complete mediation

In EROS, resources include pages of memory, nodes (fixed-size arrays of capabilities), CPU time, network connections, and anything that is built out of these. Every individual page, node, or other resource is named by one or more capabilities. Each capability names an object that is implemented by the kernel or another process in a separate address space. Capabilities are the only means of invoking operations on objects, and the only operations that can be performed with a capability are the operations authorized by that capability. This means that every resource is mediated and fully encapsulated. In most cases, a client cannot distinguish between system objects and objects that the server software implements. We can thus view an EROS system as a single large space of protected objects. Table 1 illustrates some of the key differences between capability systems and current conventional systems.

Complete accountability

Although many systems claim complete accountability as a goal, few actually implement it at the kernel level. Failures of kernel accountability commonly take two forms:

- The kernel might fail to account for kernel metadata. Mapping metadata is particularly hard to account for, because there is no direct correlation between the number of pages mapped and the amount of required metadata on most hardware.
- The kernel might account for synthesized resources rather than real resources. A process consists of two nodes. Because they are not a fundamental unit of storage, EROS does not maintain a separate quota category for processes.

In EROS, all space-consuming resources are in terms of two atomic units of storage—nodes and pages—and these are the units that are accounted for. Applications explicitly perform all object allocations, and user-level fault handlers handle page faults. This is because a new page might need to be allocated to service the page fault, and the kernel can't know the resource pool from which the new page should come.

Explicit designation

In EROS, we can trace every operation a program performs to some authorizing capability. If a line of code performs an operation that modifies an object, the capability to that object is explicitly identified in the procedure call arguments. Because this is true, there is never any ambiguity about how and why the operation was permitted, and it becomes much harder for hostile clients to entice services into misusing their authority. Even the right to execute instructions is not innate—an application that does not hold a schedule capability does not execute instructions (least authority).

Credible policy

This principle might be restated as "bad security drives out good" and is best illustrated by example. A commonly desired security policy is, "Fred shouldn't have access to this object." Unfortunately, if program A has a capability letting it speak to program B, and A also has a capability to some resource R, then A is in a position to access R on behalf of B (that is, to act as a proxy). If two programs can communicate, they can collude. An Interface Definition Language (IDL) compiler can automatically generate the code to do so. The only way to really prevent Fred's access is to isolate his programs completely from all other programs, which is generally not the policy that people want.

Because of this, EROS does not attempt to prevent the transmission of capabilities over authorized channels. Security is not achieved by preventing this copy. EROS stops programs from colluding if there is no authorized communication path between them, but its goal is to ensure that such paths cannot arise. We have yet to identify a feasible security policy that cannot be implemented this way.

Least astonishment

For the most part, we can implement the principles shown in Figure 1 without conflict. One exception is the principle of least

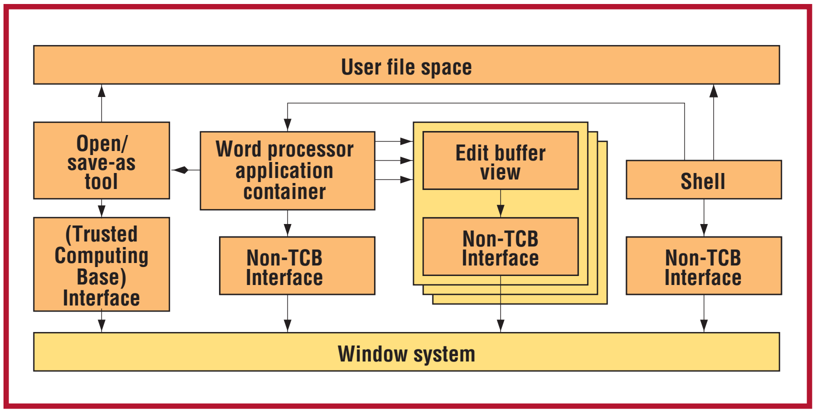


Figure 3. Components connected by capabilities.

astonishment, which is violated in the capability invocation specification. If a process specifies an undefined address as the destination of an incoming data string, the kernel will truncate the message rather than let the fault handler for that region run. The problem is that messages are unbuffered (as required by the stateless kernel principle), the fault handler is untrusted, and the process sending the message might be a shared service. A denial-of-service attack against the service can be constructed by providing a fault handler that never returns. The kernel therefore truncates the message rather than risk a denial of service.

This is astonishing to such a degree that one conference publication has cited it as a design flaw. On examination, there is a fundamental collision of principles in this area, and there are only three possible resolutions: buffering, timeouts, or truncation. Buffering violates several design principles (stateless kernel, least common mechanism, complete accountability, and no kernel allocation), and timeouts preclude repeatability under heavy load. So, given that a well-intentioned application is always in a position to provide a valid receive region, truncation appears to be the least offensive strategy for preventing denial of service.

Component-based applications

We now turn our attention to the structure of EROS applications. It is now hopefully clear that the facilities the EROS kernel directly provides are relatively low-level. Application code implements most of the system functions—even trusted functions.

For example, the EROS kernel directly provides pages of disk storage but not a file system. The file abstraction is built entirely at the application level (separation of mechanism and policy), and the file application simply stores the file content in an address space, growing the address space as necessary to hold the entire file. The file application's responsibility is to implement operations such as read and write that act on the file. The checkpoint mechanism provides stabilization. Because a distinct object implements each file, this implementation maintains the principle of least common mechanism.

This design pattern—creating higher-level functions by composing the underlying primitives of the operating system in reusable components—is the basic strategy for building EROS applications. A separate process implements each component instance, and the kernel provides a high-performance interprocess communication mechanism that enables these components to be efficiently composed. In fact, it is rare for EROS applications to manipulate kernel-supplied objects directly. Most applications reuse components that the system supplies or implement new components that provide a needed function in a structured way. This naturally leads programmers to apply the principle of least privilege in their application designs, because these components are designed to use only the capabilities they need.

Application structure

EROS applications are structured as protected, capability-connected components (see Figure 3). Each instance of a component runs with individually specified capabilities that define its authority. Capabilities are kernel protected, as are the objects they designate. The only operations that can be performed with a capability are the operations the object defines. Because of this combination of protection and mediation, an application that executes hostile code (such as a virus) cannot harm the system as a whole or other users and can't exploit the user's authority to compromise other parts of the user's environment. Similarly, capabilities control access to resources, preventing hostile code from overconsuming resources and making the rest of the system unusable.

In Figure 3, the word processor is factored into a container component and individual editing components. The container component has access to the user's file system only through a trusted "open/save-as" dialog system, but the editing components have no access to the user's file system. While the word processor has nontrusted access to the window system, the open/save-as tool has access through a special, trusted interface. The window system decorates trusted components with different window decorations, letting users know that they are interacting with a component that has potentially sensitive authority.

Testability and defense in depth

Designing applications as compositions of small components simplifies testing. Due to complete mediation, each component can be invoked only through its intended interface. Because they tend to be small and well isolated, EROS components also tend to be easily tested. A well-written test suite can typically reproduce and test all the states that are actually reachable by client code. An IDL compiler commonly generates external interfaces to components, which largely eliminates the risk of buffer overrun attacks. Because each component has a well-defined, protected interface, it is often possible to deploy new component versions into the field and test them against real applications by running them side by side with the current working version and comparing the results.

Mediated components and least privilege also make the propagation of viruses more difficult. Compromising any single component doesn't really buy the attacker very much, because the component's actions are restricted by the capabilities it can invoke. Assuming that an attacker does compromise some part of the system, he has no readily exploited communication path by which to expand the compromise. All his interactions with the rest of the system are constrained by the protocols defined at the capability boundaries. Unlike firewalls, which must operate at the network level with relatively little knowledge of the application state, the capability interfaces operate at the application level with narrowly defined component-specific interfaces. This provides the system overall with a type of "defense in depth" that is difficult (perhaps impossible) to achieve in applications that are structured as a single, undifferentiated address space.

Contrast this with current systems. Once something compromises a piece of an application, the entire application is compromised. As a result, the virus gains all the authority that the application holds—even if the original application didn't actually use that authority. A Unix-based email reader has the authority to overwrite any file that the user can overwrite. The reader doesn't do this because the program is well-behaved, but when a virus takes over the email reader, it can run any code that it wishes, usually with the full authority of the user running the application. In a capability system, this is not true.

Constructors

Closely related to the EROS component model is a generic system utility component called the constructor. When a developer writes code for a new component, she needs some mechanism to instantiate new copies of this component. This is the constructor's job. There is a distinct constructor for each type of object in the system. To instantiate an object, the client invokes a capability to its constructor.

The constructor's second, more important task is to prevent information leakage. One of the key questions that a programmer would like to ask about a component is, "If I were to create one of these components and give it some vital piece of information, could the information be disclosed without my permission?" The constructor can determine whether the component it creates is "leak free."

This is possible because all of a component's possible actions are determined by the capabilities that the component initially holds. If these initial capabilities are (transitively) read-only, then the only way the component can communicate is by using capabilities supplied by the program creating the component. Such a component is said to be *confined*.[5] As long as the creator is selective in giving capabilities to the new component, information cannot leak. Because the constructor creates the component in the first place, it is in a position to know all the capabilities that the component holds and therefore can certify the component's safety. In spite of its security features, the constructor creates new processes very quickly. In practice, we find that programmers use constructors as the generic program instantiation tool for all programs (whether or not they are confined).

Surprisingly, the "all capabilities must be transitively read-only" restriction is almost

Because they tend to be small and well isolated, EROS components also tend to be easily tested.

The use of capabilities and transparent persistence distinguish EROS from most other operating systems.

always enough. To date, the only applications we have seen that can't be straightforwardly built under this restriction are things like networking subsystems. The network subsystem needs access to external devices, and because of this, it is necessarily a potential source of information leakage. The whole point of a network, after all, is to leak information. Leaky programs aren't inherently bad, but they must be carefully examined.

The constructor is therefore the key to safely executing untrusted code. If untrusted code is executed within a confinement boundary, it can't communicate with the rest of the system at large. Although resource attacks (on the CPU or space, for example) are possible, we can restrict both the CPU time and space allocated to a confined subsystem. This means, for example, that a Web browser might be designed to instantiate a new HTML rendering component for each page. Within this component, it is perfectly safe to run untrusted scripting code, because the component as a whole is confined. The scripting code therefore does not have access to anything sensitive.

Costs and benefits

The use of capabilities and transparent persistence distinguish EROS from most other operating systems. Although component-based designs are well accepted, they require restructuring the application. Protection carries an associated performance overhead, so it is reasonable to ask what this design approach costs.

Adapting applications

Even if compatibility environments for existing applications can be constructed (a binary compatible Linux environment is in progress), EROS imposes a significant cost in development effort. To gain advantage from the underlying kernel's security properties, we must refactor critical applications into components. The most critical of these applications are external interfaces, such as SMTP, LDAP, HTTP, and FTP servers. These services run with a great deal of authority and use security-critical code. When completed, the EROS system will ship with all these services.

After servers, the next most important category is applications that execute active content such as scripting languages: browsers, email agents, and word processors. In current software, the refactoring points in these applications often already exist and are easily modified. Word processors, for example, typically open files for writing only after putting up some form of a file dialog box. Modifying the dialog box mechanism to be a protected subsystem and return an open descriptor rather than a string would go a long way toward eliminating macro viruses. Comparable protection can't be achieved by access control lists—in an access control list system, the application runs with the same authority as the user.

Trusting the user interface

The preceding discussion glosses over an important point. As a user, how do I know that I am talking to the real file dialog box? This is a trusted user interface design issue, and although work has been done on this, it isn't a simple problem. A capability-based design helps, because, for example, the window system can implement distinguished trusted and untrusted interfaces (see Figure 3) and decorate trusted windows in a user-visible way. Because capabilities are unforgeable, an untrusted application cannot contrive to appear trustworthy. In two short sentences, we have reduced the problem of application security to properly designing the file dialog and ensuring the operating system's security and trustworthiness, which is something we can solve.

In a capability system, this type of mechanism is readily enforceable. If the installer doesn't give the application trusted access to the window system, there is no way that the application can forge a trusted dialog box. Similarly, if the only access to the user file system is provided through the file dialog tool, the application has no means to bypass the user's consent when writing user files.

Performance

Current performance measurements for EROS are based on microbenchmarks.[1] The results are limited but encouraging. Process creation in EROS, for example, involves five components: the requesting application, a constructor, a process creator, a storage allocator, and the newly created components. In spite of this highly decomposed design, the EROS process creation mechanism is three times faster than the Linux fork and exec mechanism. Page faults and mapping management in EROS are over 1,000 times faster than Linux. This is not a noticeable source of delay in typical Linux applications, but it is an absolutely critical performance issue in component systems.

Because EROS does not yet have a Unix emulator, it is difficult to directly compare applications. KeyKOS included a binary-compatible Posix implementation that was directly comparable with the performance of the Mach-based Unix implementation.[6] We expect that the EROS compatibility implementation will do significantly better.

It is difficult to measure how much of the testability and performance of the EROS family is due to principles versus careful implementation and design. Probably the clearest impact of principles on the design results is from the accountability principle, because it has forced us as architects to think carefully about resource manipulation and protection.

In terms of impact, security principles run a close second. Whether EROS will ultimately be successful remains to be seen, but the EROS family has achieved something fairly unusual: a verified security architecture with a running, high-performance implementation. As a result, EROS is currently being evaluated for incorporation into various commercial consumer devices. EROS is also being evaluated for reliability-critical services such as lightweight directory access protocol implementations and Web servers.

Two anecdotal facts are encouraging indicators: it has been well over eight years since we have found an EROS kernel bug that an assertion check didn't catch. This suggests that the principle-driven design has helped us build a more reliable system by letting us check for errors effectively.

The Systems Research Laboratory at Johns Hopkins University is building a second version of EROS, restructured to support real-time and embedded applications. We anticipate seeking EAL7 assurance evaluation—the highest level currently defined—for this system under the Common Criteria process.[7]

We have also observed that programmers using EROS develop their programs in a qualitatively different way than, say, Unix developers. The system architecture encourages them to factor applications into manageable pieces, and the protection boundaries help make these pieces more testable. There are other features of the system that encourage this as well—most notably, the event-driven style of component code. EROS is the first system in the EROS/KeyKOS family that has been exposed to a significant number of programmers. It is still too early for a list of design patterns to clearly emerge.

It is striking, however, that students can master the system and build applications quickly, even though various simplifying abstractions are not provided. The greatest practical impediment to learning seems to be abandoning their Unix-based assumptions about how processes work. Often, we find that they ask questions such as, "How do I duplicate this Unix functionality?" when they can achieve their real objective more simply using the mechanisms provided.

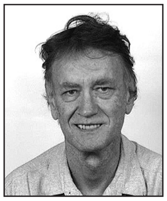
References

1. J.S. Shapiro, J.M. Smith, and D.J. Farber, "EROS: A Fast Capability System," *Proc. 17th ACM Symp. Operating Systems Principles*, ACM Press, New York, 1999, pp. 170–185.
2. N. Hardy, "The KeyKOS Architecture," *Operating Systems Rev.*, vol. 19, no. 4, Oct. 1985, pp. 8–25.
3. J.H. Saltzer and M.D. Schroeder, "The Protection of Information in Computer Systems," *Proc. IEEE*, vol. 9, no. 63, 1975, pp. 1278–1308.
4. J.S. Shapiro and S. Weber, "Verifying the EROS Confinement Mechanism," *Proc. 2000 IEEE Symp. Security and Privacy*, IEEE Press, Piscataway, N.J., 2000, pp. 166–176.
5. B.W. Lampson, "A Note on the Confinement Problem," *Comm. ACM*, vol. 16, no. 10, 1973, pp. 613–615.
6. A.C. Bomberger et al., "The KeyKOS Nanokernel Architecture," *Proc. Usenix Workshop Micro-Kernels and other Kernel Architectures*, Usenix, San Diego, 1992, pp. 95–112.
7. *Common Criteria for Information Technology Security, ISO/IS 15408*, Int'l Standards Organization, Final Committee Draft, version 2.0, 1998.

For more information on this or any other computing topic, please visit our Digital Library at http://computer.org/publications/dlib.

About the Authors

Jonathan Shapiro is an assistant professor in the Department of Computer Science at Johns Hopkins University. His research interests include computer security, operating systems, and development tools. He received a PhD in computer science from the University of Pennsylvania. Contact him at shap@cs.jhu.edu.



Norm Hardy is a senior architect at Agorics, Inc. His research interests include operating systems, security, and programming languages. He received a BS in mathematics and physics from the University of California at Berkeley. Contact him at norm@agorics.com.