An Extensible, Capability-Based Security Kernel

Jonathan Shapiro, Ph.D.
The EROS Group, LLC

April 11, 2007

Abstract

The goal of a security kernel is to permit efficient, rich but controlled communication while continuing to enforce desired security policies. In microkernel-based systems, critical parts of the system security policy are implemented by trusted application code that runs outside the nucleus. Classically conservative information flow analysis is insufficient to reason about security in these systems. Instead, it is necessary to reason about the behavior of trusted code. This reasoning must consider both the direct behavior of the trusted applications and the effects that this application has through kernel service invocations and communications with other processes. That is: the security argument must span abstraction boundaries, and must show that the argument is sustained across the composition of components. The design goal of a security microkernel is to provide a platform for application-level security enforcement.

Capability-based systems provide an ideal platform for this type of systemic reasoning, and the EROS and Coyotos projects have worked on various formal and practical aspects of this problem over the last 17 years. This paper brings these various pieces together, showing how they interact to provide an extensible security kernel: one in which we are confident that policies can be enforced while developers add new subsystems that extend the object structure of the system.

1 Introduction

After a 20 year hiatus, security kernels and formal verification have become hot topics again in the operating systems community. Whether formal verification really helps us build correct systems can be debated [4]. Rigorous specification and (human) analysis offer greater confidence in the system by exposing subtle flaws and rendering design and implementation assumptions explicit.

When we speak of a system being secure or robust, we usually mean that it maintains certain desirable properties over an extended period of time. Given this, it is striking that in most kernel specifications — including microkernel specifications — words like *invariant* do not appear. This is not always a failure of documentation. While microkernel-based design does not reduce overall system complexity, it organizes that complexity in a fashion that makes specification capture practical and invariant extraction feasible. As systems grow larger and more complex, invariant extraction becomes progressively harder. Beyond a certain point, the absence of rigorous specification is due to the fact that the intended behavior of the system is not actually known.

Even when specification extraction and capture is informal, the fact that a comprehensive specification *exists* improves the likelihood of a robust implementation. In addition to aiding security, a rigorous specification can improve performance by enabling optimizations that would be considered "too aggressive" in conventional systems. The impact of an aggressive optimization cannot be understood unless the specification of desired behavior is rigorous.

For performance, robustness, and security reasons, the Coyotos [16] family of kernels have built on both formal and informal methods to achieve and maintain rigor in our approach to system construction. KeyKOS [5] appears to have introduced the notion of an "atomic action" kernel, the EROS [13] project established that high-performance capability systems are feasible on conventional architectures, established a formal model of protected capability systems [14], and demonstrated that significant security policies can be correctly enforced by applications in capability-based systems. Coyotos brings these efforts back into commercial application in critical systems infrastructure, data center management, and medical applications. A collaborative research effort based on Coyotos explores the use of formal techniques to further improve our confidence in particular implementations of the system.

This paper brings together the key techniques used by Coyotos system in a cohesive form in order to show how capability-based systems provide a powerful substrate for secure and robust system design. We first present a basic design principle that has dictated the entire architectural history of these systems: atomic state evolution. Having described this design principle, we show how it relates directly to classical techniques of formal system modeling and analysis. We then describe how the use of a capability-based interface allows us to reason rigorously about the behavior of trusted subsystems. Today this reasoning is largely informal. We close by describing some current work that may permit us to use formal verification in the future.

2 Atomic State Evolution

Coyotos and its predecessors are built on an *atomic state evolution* (ASE) model of computation. Every path through the kernel is structured into three strictly sequential phases:

1. **Setup** Resources and locks are aquired. Objects, if

required, are made resident. Caches may be loaded or unloaded. Preconditions, including permissions, are checked. No externally observable changes to the system state are permitted (disregarding observable latency). Blocking is possible.

2. **Action** The operation requested is performed, in the sense that its semantic effects on the system state proceed. Once this phase is started, the design must ensure that the operation runs to completion without blocking. Errors discovered during this phase induce a system halt.

3. **Result** The result of the operation is returned to the invokee, which may or may not be the same as the invoker. This phase is distinguished primarily because some operations may alter the state of the invokee during their action phase, so there are corner cases to consider.

Neither resources nor kernel stack are retained by blocked processes. On wakeup, the system call is resumed from scratch. This is possible because no semantically observable changes are permitted to happen while it is still possible for a process to block.

Ford [3] *et al.* describes two common models of microkernel design: the *process model*, in which every process has a per-process kernel stack, and the *interrupt model*, in which preempted processes save a continuation and give up their kernel stack. Atomic state evolution is not quite either of these models. In an ASE design, the kernel stack is per-CPU rather than per-process. No continuation needs to be formed when a process is blocked in the kernel, because a blocked process does not retain ownership of resources.

While the temporal properties of ASE are difficult to state precisely, the Coyotos implementation includes the requirement that all kernel operations that run to completion must do so in a total number of steps that has a small constant bound. Unbounded recursion or iteration is not permitted within the kernel. The kernel is demonstrably deadlock-free. Because blocked processes in conventional kernels retain their stack, the least powerful computational model that can describe their behavior is concurrent pushdown automata. Coyotos processes do not retain a kernel stack. This, coupled with the bounded recursion requirement, means that the behavior of the kernel may be modeled as a sequential finite state machine. This simplifies reasoning about the kernel's behavior in either formal or informal contexts.

A final property of ASE is that kernel invocations are total functions over the domain of possible invocation argument payloads. At the capability invocation trap interface there is a large but finite set of possible argument payloads. While the current draft Coyotos specification is not yet sufficiently precise about these error results, every kernel-implemented capability defines an outcome for every possible argument payload (often a RequestError exception).

2.1 Origins and Implications

The ASE design approach emerged from Norm Hardy's experiences as a member of IBM's Advanced Computing Systems group, which was responsible for both the 801 RISC processor and the earliest versions of the System/360. Both processors were anchored in the notions of sequential operational pipelines characterized by cleanly separated units of operation and atomic (all or nothing) instruction effect.[1] Atomic instruction effect greatly simplified the control structure of both processors, in particular the mechanism for exception handling. Later, it provided essential underpinnings for the introduction of superscalar processing techniques [17]. Today, this design discipline is nearly universal in microprocessor design. Among general-purpose microkernels it appears to be unique to the Coyotos family of kernels.

Implicit in the atomic state evolution model is the assumption that resource re-aquisition is inexpensive. This demands that kernel paths be both short and regular. Further, the design model assumes that resource de-aquisition is inexpensive. The approach becomes impractical if abandoning a kernel path requires any significant cleanup. A superscalar processor abandons unreachable computation by un-naming the destination registers and suppressing exceptions; no large-scale cleanup is required. In a similar way, a blocking Coyotos process de-aquires its resources by incrementing a counter and branching to a well-known assembly entry point that re-sets the per-CPU kernel stack pointer to create an empty kernel stack. The counter increment is sufficient to release all objects that have been pinned in residence for the duration of the current operation. In low-frequency cases, a single newly allocated data structure must be returned to the free pool prior to blocking.

Like its predecessors, Coyotos is an object-based systems. Operations are performed by invoking protected capabilities [2] that name objects. The kernel is responsible for marshalling, transferring, and demarshalling the arguments of the invocation, but invocation payloads are completely regular. The meaning of a capability invocation is determined by the object designated by the capability. The implementor of an object may be either the kernel or an application; the set of objects implemented by the Coyotos kernel is restricted to core kernel abstractions (Processes, Pages, Guarded Page Tables), and the operations on these objects are individually very simple. Conceptually, the invocation handler procedure for each kernel object may be viewed as a functional unit of a software-defined microprocessor. As with hardware functional units, there is a well-defined point in every control flow path through these method handler procedures where the functional unit becomes committed to producing a result (which may be an exception reply message). We refer to this as the *commit point*.

The atomic state evolution approach works very naturally in microkernels whose system call API is designed with atomicity in mind. In monolithic systems, where kernel invocations involve larger resource sets and longer control flow paths, the atomic evolution model becomes difficult to implement.

2.2 Informal State Model

The informal kernel state model of the Coyotos system involves only five object types: processes, guarded page tables (GPTs) [9], pages, capability pages, and endpoints. Each of

[1] With the notable exception of the S/360 block move unit, whose behavior can be subdivided into atomic units of operation.

these objects has data "slots" and capability "slots". A process object, for example, has 32 slots that serve as its capability "registers." GPTs contain 16 capability slots that serve as the software equivalent of page table entries. Each object has one or more associated capabilities. Each capability conveys to its invoker the authority necessary to access and modify that object according to the kernel's operational semantics and the methods permitted for that capability type.

The strict partitioning of data and capability slots ensures the protection of capabilites. Applications invoke capabilities by indirect designation, as in "invoke the capability in capability register 5 and request that the following method be performed." There is a kernel-implemented capability ("KeyBits") that returns the canonical representation of an arbitrary argument capability as data. This capability is said to be "closely held," meaning that only a small list of trusted applications hold it, and these applications neither disclose the KeyBits capability nor its output. There is no kernel operation that converts data into capabilities. In consequence, capabilities cannot be "leaked" through covert channels. Even if the bits of the capability representation are known, the recipient cannot use them to re-construct the actual capability.

System state evolves as the result of capability invocations. Typically, a capability invocation modifies a three objects: the invoker (whose program counter, at a minimum, must be advanced), the object designated by the invoked capability, and the invokee that receives the response to the invocation (which may or may not be the invoker). Because kernel invocations are conceptually atomic, the current state of the system can be characterized inductively as the result of a sequence of indivisible steps applied to some known initial state.

With luck, these steps are correctness preserving with respect to the system's operational semantics. The atomicity rules help to ensure this by localizing all mutation of system state into a very short sequence of "action" code that is easily inspected. While the setup phase of an invocation may alter cache state, these alterations are both simply implemented, heavily exercised, and consistency checkable (more on this below). The result phase modifies the state of the receiver, but there are few boundary cases at this stage of a kernel operation, and the same result delivery code is used for all kernel value returns. As with the setup phase, this code is heavily exercised and tested.

2.3 Role of Checkpoint

Periodically, the kernel stops serving system calls briefly in order to perform a "snapshot" operation. This operation efficiently marks all kernel state copy-on-write. Execution resumes while the captured snapshot is asynchronously written to a transaction log.

The checkpoint mechanism has been described in detail elsewhere [8, 15] From a robustness and security perspective, the important point about checkpoint is that the system state induction is preserved across restarts. Each snapshot constitutes a consistent cut of the system state that is written to disk as a single unit. This is in marked contrast to process checkpointing mechanisms or non-checkpointing systems, where no consistent cut mechanism exists. We know of know techniques for reasoning about the consistency of a system state derived from an inconsistent cut.

3 Operational Checks

The Coyotos implementation borrows a range of mechanisms from the earlier EROS system to ensure that kernel operations preserve correctness:

1. Immutable objects are guarded by a software-implemented in-memory checksum. If the checksum fails, the object is invalidated.

2. Kernel objects are similarly guarded by a checksum. This checksum is updated whenever the kernel object is modified.

3. The kernel caches object state in various ways. For example, the state of GPTs is cached in page tables to allow load and store instructions to proceed. As a consequence of this caching, dependency tracking structures are created. Checkable consistency invariants can be stated that describe the legal relationships between the cached state, the dependency tracking entries, and the "official" state.

These simple mechanisms do a surprisingly good job of protecting the system from run-time errors. The kernel runs continuous incremental consistency checking as a background task, validating that all object states match their checksum, and that state which is multiply represented is self-consistent. During development, we have seen several cases where pointer errors have damaged various data structures. Approximately 90% of all kernel state is covered by consistency checks, with the consequence that these errors are almost always caught by the background consistency checked before a checkpoint can be committed. By the standards of kernel debugging, localizing these errors is surprisingly straightford. A full consistency check can be run on each kernel entry and exit to isolate the offending call, and binary subdivision can be used on the bad control flow path to quickly isolate the source of the error.

Because of these consistency mechanisms, our experience in EROS was that *bugs* were not the major source of errors in the Coyotos implementation. The challenging problems were due to failures of specification or failure to adhere rigorously to the requirements of the ASE model. The Coyotos specification [16] is significantly more complete than the EROS specification. Our work using the MOPS static checker [1] suggests that a more powerful model checker would be able to capture and check many of the kernel implementation sequencing invariants. With these checks accomplished, the kernel remains far from formally verified, but it is a relatively high confidence implementation. With this in place, we turn our attention to the problem of enforcing security policies from application-level code.

4 Conservative Safety Analysis

The classic method for reasoning about the flow of information and access rights is a graph evolution formulation. The problem formulation given by Harrison, Ruzzo, and Ullman [6] is particularly clear. The analysis proceeds by first defining the system state, and then describing the actions (HRU's "commands") that update the state. Each action is assumed to perform an atomic update. Each action has a set of preconditions that determine whether the action can be performed against the current system state. The analysis assumes that all processes will be maximally hostile. Proceeding from a known initial state, actions are performed in arbitrary order until an unsafe state is discovered or a decidably safe fixpoint is reached. A key point made by HRU is that for most real protection systems the state evolution has no least upper bound, and consequently does not converge. While the HRU decision procedure is not decidable in general, it is decidable in all finite systems, and it is decidable in all classical capability systems (finite or otherwise).

The confinement verification for EROS [14] adopts the HRU approach directly to model the behavior of protected capability systems. It defines a system state consisting of processes and objects. In this state model, objects are abstracted as infinite sets of capabilities. An abstract operational semantics captures the notion of capability updates within the graph. This operational semantics is general in the sense that every real operation of the EROS system can be directly modeled as a sequence of operations in the model. It is also general in the sense that the model describes the behavior of every protected capability system known to us. Using this model, we have built a verification proof showing the correctness of a decision procedure that tests the confinement property [7].

The HRU analysis is an example of authority analysis. Starting from the operations that are directly permitted by some initial system state (the "permissions" of the initial state), the method examines the total set of operations that might transitively be performed (the "authority" of the initial state). The *problem* with this style of analysis lies in the fact that all processes are assumed to be maximally hostile. This excludes from analysis the possibility that some processes may be responsible for *implementing* the system security policy. The conventional "solution" to trusted processes is to incorporate their behavior directly into the operational semantics of the system, effectively merging them into the kernel. In microkernel-based systems this approach is unattractive, both because it obscures the relationship between the model and the implementation, and because changing a pluggable policy module may demand a *de novo* re-verification of every system security property. In order to analyze component systems, the model must be enhanced. In particular, we must re-introduce the notion of designation.

The missing link in the method becomes apparent when we note that in the EROS and Coyotos system, the confinement decision procedure is executed by an application program (the "constructor"). This application code is trusted, and it runs an algorithm that is small enough, in principle, to be formally verified. Imagine, for a moment, that we have performed such software verification. The end result of this verification will be a statement of the form "There exists a family of capability invocation sequences X, Y, ... Z that implement the desired decision procedure. The capability invocation sequences executed by the program fall within this family, therefore the program implements the decision procedure correctly."

The modeling problem here is that the correctness of the constructor relies on invoking specific capabilities with specific arguments in specific sequence, and further relies on *not* invoking certain *other* capabilities in ways that inappropriately disclose the wrong kinds of information. That is: the constructor implements a kind of a reference monitor, and the system relies on this program to behave correctly. Such a program cannot be analyzed successfully using the "maximally hostile program" hypothesis. What we need instead is a model in which program A performs a known sequence of invocations while a second program B performs a maximally hostile sequence. We need to show that under these conditions scenario program A's implementation of its algorithm remains faithful.

5 Naming and Extensibility

In order to support the analysis of trusted programs, we need to replace the abstraction of objects as infinite sets of capabilities with an abstraction of objects as indexable vectors of capabilities. Instead of saying at each step that the hostile process invoked an arbitrarily selected capability, we now say that that the hostile process invokes a capability at an arbitrarily selected indexable location in the vector. The difference is that we can now speak about a known program that invokes a specific sequence of capabilities with known arguments.

The introduction of explicit designation (indexing) corresponds exactly to what happens in real capability systems: the application performs an *invoke capability* system call in which an index for the invoked capability is presented. Once the indexing mechanism is captured by the formal model, the operational semantics of the state graph can be related to the execution semantics of the trusted program. That is, we can now perform a three-step analysis of the form:

1. If a set of preconditions can be established, some desired security policy is known to be enforced.

2. There exists a particular sequence of capability invocations that confirms these preconditions.

3. The trusted program actually performs that sequence of capability invocations.

The key enabler of this is that capabilities allow us to establish a direct relationship between the procedure calls of the program that invoke capabilities and the objects and operations of the system state model. It is not an accident that the paper introducing the term "capability" is entitled *Programming Semantics for Multiprogrammed Computations* [2].

6 Application-Enforced Policies

The Coyotos kernel implements capability-based permission checks, but it does not in itself enforce any particular security policy. For example, the kernel implements a capability *range* that permits fabrication of arbitrary object capabilities and destruction of arbitrary objects. Any holder of this capability is in a position to violate any security policy one might imagine. That is: certain system configurations are insecure.

The safety and security of Coyotos in the presence of *range* relies on an argument in several parts:

- The *range* capability is held only by a trusted application: the system storage allocator.
- The *range* capability is closely held. The system storage allocator does not expose it directly or allow it to be misused.
- The storage allocator algorithms ensure that all invocations of the storage allocator make safe use of the underlying *range* capability.
- Therefore, a system configuration consisting of a *range* capability that is closely held by the storage allocator is safe.

The key points here are that (a) it is the storage allocator application rather than the kernel that is imposing the security policy (via its safety checks), and (b) it is the explicit designation of capabilities (therefore permissions) at the kernel invocation interface that makes this possible. In particular, we can imagine implementing a program having two clients. Each client stores a capability in this program that must not be disclosed to the other. By coupling program correctness verification techniques with explicit designation, we can confirm (or disprove) that the program acts correctly.

In Coyotos, many key security constraints are enforced through this combination of explicit designation and system configurationd. Certain capabilities are closely held. The confinement test is implemented by a (trusted) application. Rajunas *et al.* [12] demonstrated a working multilevel security design implemented entirely at application level that operates correctly in a system that simultaneously implements unrelated security policies over other portions of the overall system.

There are two senses in which the Coyotos "security kernel" is extensible:

1. New objects with new permissions models can be introduced. Reference monitors that do not trust the implementors of these objects can ensure (by interposition) that these objects cannot be invoked by sensitive programs.
2. Many policies can be implemented simultaneously in a single system image. While there is no known solution for arbitrary composition of security policies, hierarchical arrangements of information flow policies are both straightforward and useful.

7 Related Work

Numerous papers have proven that capability-based security is for one reason or another impossible. In most cases these papers arrive at their results through failed analysis. They ask what capabilities as a permissions structure can enforce directly, as opposed to examining how application-level trusted code can built on capabilities to enforce policies of interest. As we have illustrated here, capability systems are both extensible and compositional. New objects can be introduced, and these objects can enforcably impose new security policies if the system configuration is appropriately constructed.

Neumann *et al.* [10] explored a significant portion of this design space, including a significant thrust of verification, in their work on the Portably Secure Operating System. Their retrospective paper gives a cohesive overview of their approach [11].

8 Conclusion

The role of a security microkernel is to provide an effective substrate for application-level definition and enforcement of security policies. Reasoning about system configurations requires that the actions of these trusted programs be coupled to the operational semantics of the microkernel in order to understand their impact on authority. Because the capability graph update model is decidably analyzable, and because the explicit designation provided by capabilities offers the type of semantic coupling that is required for trusted program analysis, capability-based systems are ideally suited for this type of reasoning.

While the Coyotos kernel is not formally verified, there is a rigorous correspondence between the implementation and a formal system model. This correspondence has been partially checked using automated tools [1], which provides confidence that the model requirements are upheld in the implementation. The level of confidence obtained is sufficient for security monitoring and critical data center management applications. We are currently evaluating whether more complete analysis is warranted for surgical control applications. Given the currently observed reliability level of the system, it is not clear how to strike an appropriate balance between improvements in confidence and cost of verification. In particular, it is not clear whether a greater improvement in reliability might not be had if the cost of a formal verification were instead spent on more thorough human analysis and/or greater development effort.

Our current research work is proceeding on two complementary tasks. The first is a machine automation of the confinement verification proof, replacing sets with indexed maps. The re-verification of confinement in the automated framework is nearing completion. The second thrust is BitC, a programming language that is designed to let us write efficient "systems" programs will still being able to do more extensive automation of program property checking.

References

[1] H. Chen and J. S. Shapiro. Using Build-Integrated Static Checking to Preserve Correctness Invariants. *Proc. 2004 ACM Symposium on Computer and Communications Security*. Oct. 2004.

[2] J. B. Dennis and E. C. van Horn. "Programming Semantics for Multiprogrammed Computations" *Communications of the ACM*. **9**(3), March 1966. pp. 143–154.

[3] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann", "Interface and Execution Models in the Fluke Kernel," *Proc. 3rd Symposium on Operating System Design and Implementation*, Feb 1999, pp. 101–115.

[4] P. Gutmann, *The Design and Verification of a Cryptographic Security Architecture*, Ph.D. Thesis, University of Auckland, New Zealand, 2000.

[5] N. Hardy. "The KeyKOS Architecture." *Operating Systems Review* **4**(19), Oct. 1985, pp. 8–25.

[6] M. A. Harrison, W. L. Ruzzo, and Jeffrey D. Ullman. "Protection in Operating Systems." *Communications of the ACM*, **19**(8), pp. 461–471. Aug 1976.

[7] B. W. Lampson.: A Note on the Confinement Problem. *Comm. ACM*. **16**(10), 1973, pp. 613–615.

[8] C. R. Landau, "The Checkpoint Mechanism in KeyKOS." *Proc. Second International Workshop on Object Orientation in Operating Systems*. Sep. 1992, pp. 86–91, IEEE.

[9] J. Liedtke. *On the Realization of Huge Sparsely-Occupied and Fine-Grained Address Spaces*, Dissertation, Technische Unversität Berlin, 1996.

[10] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson: *A Provably Secure Operating System: The System, Its Applications, and Proofs*. Computer Science Laboratory Technical Report CSL-116, 2nd ed., May 1980, SRI International.

[11] P. G. Neumann and R. J. Feiertag. "PSOS Revisited." *Proc. 19th Annual Computer Security Applications Conference (ACSAC 2003)*. 2003

[12] S. A. Rajunas. *The KeyKOS/KeySAFE System Design* Tehnical Report SEC009-01, Key Logic, Inc., March 1989.

[13] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS, A Fast Capability System. *Proc. 17th ACM Symposium on Operating Systems Principles*. Dec 1999, pp. 170–185. Kiawah Island Resort, SC, USA.

[14] J. S. Shapiro and S. Weber. Verifying the EROS Confinement Mechanism. *Proc. 2000 IEEE Symposium on Security and Privacy*. May 2000. pp. 166–176. Oakland, CA, USA

[15] J. S. Shapiro. "Design Evolution of the EROS Single-Level Store." *Proc. 2002 USENIX Annual Technical Conference*. USENIX Association, 2002.

[16] J. S. Shapiro, J. W. Adams, E. Northup, M. S. Doerrie, N. H. Walfield, and M. Brinkmann, *Coyotos Microkernel Specification*, April 2007.

[17] R. M. Tomasulo, "An Efficient Algorithm for Exploring Multiple Arithmetic Units," *IBM Journal of Research and Development*, **11**(1), Jan 1967, pp. 25–33; IBM.