Towards a Verified, General-Purpose Operating System Kernel[†]

Jonathan Shapiro, Ph.D., Michael Scott Doerrie, Eric Northup, Swaroop Sridhar, and Mark Miller

Systems Research Laboratory
Department of Computer Science
Johns Hopkins University

Abstract. Operating system kernels are complex, critical, and difficult to test systems. The imperative nature of operating system implementations, the programming languages chosen, and the usually selected implementation style combine to make verification of a general-purpose operating system kernel impractical. While security policies have been verified against *models* of general-purpose operating systems, no verification has ever been accomplished for a general purpose operating system *implementation*.

This paper summarizes how we are attempting to create a verified general purpose operating system implementation for **Coyotos**, the successor to the EROS system, and why we believe that there is a reasonable chance of success.

Introduction

The current state of affairs in computer security and reliability is unsupportable. We *must* find ways to build software systems that are robust and survivable, and develop techniques and tools that can bring these development practices into mainstream product development. The problem is foundational: there exists, in principle, no evolutionary path from current operating systems technology to a secure or survivable alternative. Without an operating system on which applications can rely, secure applications and defensible systems are impossible to build.

The only technique currently known that will allow us to build an operating system of the required robustness is formal verification. Verification has been successfully applied to various special-purpose critical software systems (most notably critical flight control software), and it has become a key part of commercial microprocessor development, but it has not been successfully applied to a general purpose operating system kernel. There are several reasons for this:

- Few operating system designs incorporate a rigorous notion of what constitutes a "correct" or "consistent" state of the system.
- Few people who write operating systems understand verification.
- Current systems programming languages have no formally defined semantics, and suffer from problematic ambiguities.
- Most operating systems have no clearly identifiable "unit of operation" boundaries in the execution of the system where the system state is (alleged to be) consistent. This makes correctness verification difficult or impossible.
- Most kernels use non-preemptive multithreading within the kernel.[1] Even on single processor systems, multithreading creates an exponential explosion of the state space that the prover must consider — far beyond what is currently feasible to verify.
- Imperative programming languages *also* create an exponential explosion of the state space that the prover must consider.
- Provers, with the notable exception of ACL2, might fairly be said to present a "programmer hostile" interface. The language of expression used by the prover needlessly departs from the language used by the programmer, imposing a significant conceptual translation burden on the developer.
- Projects that contemplate the use of verification tools often relegate responsibility for verification to a side team in order to relieve programmers of the "burden" of verification. One result is systems that cannot be verified because their authors don't know how.

Given these issues as initial conditions, it is understandable that verification is not a high priority for operating system developers in the wild. In spite of this, there are at least three results which suggest that verifying a suitably structured microkernel system may now be feasible:

PSOS (1980) While the system was never completed, a substantial framework for verification was crafted for the PSOS system. This work had heavy influence on the subsequent evolution of *nqthm* and later *ACL2*.

See: *A Provably Secure Operating System: The System, Its Applications, and Proofs* [9].

KIT (1989) Bevier's verification of the KIT kernel against a simple microprocessor model is of approximately the same order of complexity as the verification required for a modern microkernel.

See: *Kit: A Study in Operating System Verification* [2].

VLISP (1995) The VLISP project's successful verification of the pre-scheme compiler and runtime system similarly suggests that programs of the size

[1] The UNIX kernel's `sleep()` call, for example, typically causes a context switch into a different kernel continuation.

and complexity of modern microkernels should be "within reach" of modern automated provers, provided they can be implemented in a suitable language.

See: *The VLISP Verified Scheme System* [5].

The Hopkins *Systems Research Laboratory* is starting work on the **Coyotos** kernel, a successor to the EROS system [13]. As part of this, we are trying to achieve a verified implementation of the kernel and the system's key utilities. We are pursuing this for several reasons:

- It is an intrinsically interesting research challenge.
- We are attempting to build a system that exceeds the requirements for EAL7 evaluation under the *Common Criteria* evaluation scheme. We believe that a fully verified correspondence argument for the implementation is easier to achieve, more rigorous, and easier to maintain than the semi-formal correspondence required for EAL7 evaluation.
- An "open proofs" system (one in which both the system code and the verification of correctness are public) would serve as a public example of how to go about building a robust, secure system. It would allow customers to hire independent experts to validate the verification. It would allow experimenters to attempt changes to code and proof as a learning vehicle.
- An open source, open proofs demonstration that verified systems are possible may fundamentally change both user expectations about critical systems and the "standard of diligence" that must be established to sustain claims of non-liability for critical system software flaws.
- We don't see any other way to get to long-term survivable software systems, especially for critical infrastructure.

For kernels and similar critical systems, we need to know that all operations terminate in a (tightly) bounded number of steps. This means that verification must be concerned with establishing *total correctness* properties. The EROS system is unusual in having a rigorous notion of consistency, an existing formal system model (with a successful paper verification [14]), and a well-defined notion of "unit of operation" (it is an interrupt-style kernel). Bounded time operations, and therefore termination, were a specific and pervasive concern in the EROS design (and its predecessors). Every invocation on the EROS kernel honors the ACID properties, which allows us to express system call semantics as atomic, consistency-preserving transformations on a well-defined system state. Coyotos, the EROS successor, retains these properties and significantly reduces both the semantic and implementation complexity of the kernel.

A key problem we face is the problem of programming language. There exist languages such as ML that are strongly typed and formally specified. While considerable work would be required, it is in principle straightforward to take an approach similar to that of ACL2 [7]: capture a full semantics for an ML language subset (notably exluding the module system) in an automated prover, and reason

about programs written in this subset. Unfortunately, ML and similar languages have several key limitations from the perspective of kernel development:

- They do not provide machine-level, fixed size representation types.
- They provide insufficient control over low-level data layout. In particular, systems codes require the ability to specify both unboxed composite types and unboxed references. This is both a performance and a correctness issue; the layout of certain data structures is dictated by the underlying hardware.
- The incorporation of full tail recursion in the language specifications means that high-performance compiler implementations cannot exploit C as a structured assembly language [1][16]. This significantly increases the cost of implementing a suitably modified subset of these languages. Fortunately, full tail recursion is not a real-world requirement. In practice, a more constrained form of tail recursion is probably sufficient.
- Most safe languages rely intensively on dynamic memory allocation. In some cases this reliance is embedded so deeply that it is impossible to write programs that do *not* allocate memory dynamically. Kernels must be capable of operating with predictable variance in a fixed-memory environment. Dynamic allocation renders this problematic.
- With the exception of ACL2, no existing language provides means to integrate theorems and their proofs into the body of the program. From an assurance and robustness perspective, these meta-statements about the program are as important as the program itself.

In light of this, a key challenge for the Coyotos effort will be defining a programming language whose unambiguous semantics can be formally specified in mechanical form, is capable of capturing the efficiencies of low-level representation, and can be successfully used by hardcore systems programmers.

The balance of this paper briefly highlights some relevant attributes of the EROS system architecture, our current plans for the evolution to Coyotos, our approach to building an implementation language, and some of the properties that we would ultimately like to verify.

EROS

EROS is a high-performance, capability-based operating system that runs on conventional microprocessors [13]. It minimally requires a processor that provides paged memory management hardware and a reliable separation between user and supervisor execution. The current version of EROS executes on the Pentium processor family. The predecessor system, KeyKOS [4] has been ported to the Motorolla 88000, the IBM System/360, and the Sun SPARC processor families.

Along with the L4 system [11], EROS stands as one of two major remaining microkernel-based research systems. Where L4 has historically focused on sys-

temic performance issues in microkernel-based systems, EROS has focused primarily on security. Where L4 has shown that microkernel-based systems can be fast, EROS has shown that they can also be protected without sacrificing performance.[2]

System Model

The architectural model of EROS is that the kernel provides a protected extension of the underlying microprocessor, augmenting the hardware features with support for kernel-protected capabilities and implementing a canonical interface to the system's memory mapping and exception handling mechanisms using capabilities as the fundamental protection mechanism.

One may view the execution of an EROS system as the steps of a sequential state machine whose transitions consist of:

- Execution of a single, user-mode machine instruction, *or*
- Delivery of an exception notification to a user-level fault handler via IPC, *or*
- Execution of an application-initiated "invoke capability" exception, *or*
- Processing of some pending interrupt event, which may cause a preemption of the current user process.

This view of EROS in terms of an explicit operational semantics is foundational in the EROS security model. The execution of an EROS system begins in a hand-constructed consistent state, and the continued security of the system rests on an inductive argument that every instance of the operations identified above performs an atomic, consistency-preserving transformation on the global system state. To support the logic of this argument, EROS is transparently persistent. Every few minutes, an instantaneous global checkpoint is taken of the entire system state. This snapshot is then incrementally written to disk as execution proceeds. When the system powers up, it resumes execution from the most recently saved checkpoint image.

A key underlying aspect of this model is that EROS kernel invocations are atomic. Every kernel invocation (including IPC) proceeds in two phases:

Prepare During the prepare phase, all required resources are determined to be in memory and are pinned in memory for the duration of the current operation. If an object is to be mutated by the current operation, the prepare phase reserves sufficient space in the system checkpoint area to hold the modified version of the object.

[2] This characterization is not entirely fair. The L4 effort has pursued a number of areas, including real time systems and control of systemic performance, that have not been addresed by the EROS effort.

Action During the action phase, the requested operation is performed. By both design and requirement, the action phase is not permitted to fail. More precisely, the only form of failure permitted during the action phase is to halt the machine. This may occur, for example, if memory is discovered to have an ECC error. During the action phase, the process is not permitted to block, and the kernel is obligated to execute the current invocation to completion.

During the prepare phase, no "semantically observable" modification to the system state is permitted. Changes to kernel caches, rewriting of internal kernel data structures into alternative representations, and queuing of invoking processes on event completion queues in the kernel are not considered to be semantically observable events.[3]

EROS is an interrupt-style kernel. In the event that some action occurs during the prepare phase that might violate a correctness precondition previously established during the prepare phase, the current system call is restarted from scratch. No kernel stack is ever retained by a blocked process. Because no semantically observable mutations have been allowed during the prepare phase, this "abandon and restart" policy is always safe (though it does introduce proof obligations concerning liveness properties).

At some well-defined point on every static control flow path in the kernel, there is a conceptual boundary line that we refer to as the "commit point." This line marks the transitional control point between the prepare phase and the action phase. In the current kernel implementation, there is an explicit call to an inlined, empty procedure at every commit point. This allows us to use static control flow model checking to verify both that semantically observable mutations occur only after the commit point and that no process performs any action after the commit point that might block.

Finally, there is a global design requirement that every kernel path must complete in O(1) steps — that is, within a known constant number of instructions. In fact, we require that this be a (somewhat fuzzily expressed) small constant bound. Indeed, one of the significant changes between the earlier KeyKOS system and the current EROS design was the elimination of the last kernel operation that lacked a small constant time bound.

Though neither the KeyKOS nor the EROS designers realized it at the time, both groups informally but rigorously introduced measure conjectures into the system implementations. With the benefit of deeper hindsight, *all* of the recursive and iterative algorithms of the EROS kernel have straightforwardly stated and verifiable measure conjectures. KeyKOS, with the exception of a single scheduling-related algorithm, also had this property.[4]

[3] Process en-queuing *is* observable in the form of latency, but this type of observation has no effect on the overt security properties of the machine.

[4] In KeyKOS, the flush algorithm for the meter tree could hypothetically visit every meter node in main memory. While this visitation is bounded by the size of memory,

Without exception, every operation performed by an EROS application, including the execution of non-privileged instructions, may be expressed as a capability invocation. For normal instructions, the process is implicitly invoking a process capability to itself in order to rewrite its register state. For kernel calls, the capability invoked is directly identified in the invocation. For memory operations, the capability to the object ultimately manipulated (the page) is reachable by traversing a path beginning from the per-process address space capability, and computing the path access rights as an aggregation of the stepwise permissions granted by each capability in the path. Finally, exceptions may be modeled as an invocation of a capability to the appropriate fault handler. In consequence, the permission to perform any action is straightforwardly defined, easily checked, and conveniently accumulated during the traversal of a referencing path that needs to be traversed in any case to locate the target object of the operation.

In most capability systems, the permission accumulation rule is to begin with maximal permission and compute the intersection of these initial permissions with the stepwise permissions as the path is traversed. In the EROS system, the *weak* access restriction somewhat complicates this rule.

The weak access restriction provides a "transitive read-only" permission. There is (transitively) no way to obtain any capability that conveys mutate authority by proceeding from a weak capability. In practice, this restriction is performed stepwise: fetching a capability (even within the kernel) from an object named by a weak capability returns a weakened variant of the fetched capability: one whose access restrictions include both *read only* and *weak*. In some cases, this downgrade is performed conservatively by returning an invalid capability. Because the next capability at each step in the path traversal is conditionally transformed based on the permissions of the currently traversed path prefix, the simple accumulation rule must be replaced by fusing the accumulation of permissions into the operational definition of path traversal. This fusion proves to be useful, as it helps to reduce the possibility of a discrepancy between the traversals *performed* by the machine and the traversals *permitted* by the machine.

The weak access right is *not* essential from the standpoint of expressive power. A system without it can be constructed in such a way as to preserve overt confinement and partitioning. The weak access right *is* essential from the standpoint of resource efficiency and performance. Using the weak access right, for example, it becomes possible for two processes to share access in copy-on-write form to a common read-only graph of objects, even if the shared graph contains internal, write-authorizing capability references. This proves to be a significant enabler for some common microkernel design patterns, most notably the use of user-defined memory fault handlers. The weak access right also significantly reduces

---

and a bounding measure conjecture can therefore be stated for it, it does not satisfy the "small constant bound" design objective shared by the two systems.

the number of operations that must be monitored and interposed by a reference monitor to implement mandatory access control policies.

The EROS confinement mechanism is constructed on top of the weak right. Lampson defines confinement as inability to communicate over unauthorized channels [8]. The EROS constructor mechanism enforces *overt* confinement (i.e. confinement ignoring covert channels). While this mechanism does not completely satisfy the Lampson definition, the enforcement of covert channel restrictions is largely an orthogonal problem. In EROS, a process is overtly confined *iff* it can be shown that all of its authority to mutate originated with capabilities provided by the instantiating client. That is, all of the *initial capabilities* held by the program instance at instantiation time are (transitively) immutable. This test is implemented by a user-mode, trusted application: the *constructor*. The constructor performs a static test prior to instantiation to validate that all of the immediate initial capabilities (as opposed to those that are transitively reachable from these) are either:

- Trivially safe kernel-implemented capabilities, *or*
- Weak (therefore transitively immutable), *or*
- Capabilities to another constructor that in turn certifies its instantiations as confined. This is acceptable because the constructor is trusted code and one constructor is able to authenticate another. This case provides an inductive extension of the previous two rules, and enables instantiation of complex confined subsystems with rich behavior.

It has been demonstrated in the KeySafe [10] system that the constructor provides a sufficient foundational mechanism to implement mandatory access controls such as multilevel security. Of perhaps greater pragmatic importance, pervasive use of the constructor as a process instantiation mechanism provides a foundation for defense in depth, as demonstrated in the EROS network stack [12] and the EROS trusted window system [15].

Resource Allocation

If we are to reason about the total correctness of a system, resource allocation is a critical concern. In order to return a correct result, a process must have sufficient space and compute time. This introduces a proof obligation concerning resource allocation that must be discharged. Determining resource sufficiency is possible if the maximal resource requirements of all processes are fully known and the system is sufficiently provisioned. This specialized solution can be extended using temporal non-interference reasoning to further cases, but in general the resource sufficiency problem is intractible.

From the kernel perspective, it is necessary either to reason explicitly about resource allocation or to somehow *avoid* such reasoning. EROS takes the latter approach by eliminating kernel resource allocation altogether. The kernel is

responsible for the *safety* of kernel resources, but it is not responsible for the *allocation* of these resources. Responsibility for resource allocation is delegated to (trusted) application level code.

In EROS, this property is slighty relaxed by allowing the kernel to cache protected state for performance reasons. In some sense, this form of caching multiplexes a fixed resource over unbounded usage demand, but the kernel design ensures that every cache can either be discarded without observable semantic consequence (again barring latency) or written back into some definitive representation object that was allocated by a user-level allocator. The end result is that the kernel is entirely deadlock-free.

This design approach extends to address space mapping structures as well. In EROS, the address space of a process is defined by explicitly user-allocated data structures called *nodes*. The hardware mapping tables are constructed by the kernel on demand by traversing the node structures, which are the definitive statement of the mapping. The hardware structures are managed as a discardable cache. In addition to its role in address space definition, the EROS node structure is also used as the persistent representation of process state.

The EROS address space definition approach is in contrast to the L4 *map* operation, which implicitly allocates a kernel mapping database node. The difficulty with the mapping database node is not that it is implicitly allocated,[5] but that its state is definitive and unaccounted. If the mapping database node is discarded, it is not always possible to reconstruct the mappings that depended on that mapping database node. This induces restrictions on application use of the L4 map operation in order to ensure that the mapping database nodes *can* be discarded. To our knowledge, no current L4 implementation treats the mapping database as a cache, and the practical design implications of such treatment have not been explored in current L4-based systems.

From EROS to Coyotos

Coyotos is the successor to the EROS system. While EROS has satisfied most of our research objectives, the system suffers from several practical impairments:

- Though it simplifies security reasoning, transparent persistence is not cleanly compatible with translucent network operations.

  Persistence is removed in Coyotos.
- The EROS *node* data structure, which was introduced to support persistence, complicates both the implementation and the specification of the system:

[5] The *map* operation can be implemented in such a way that every *map* invocation allocates exactly one mapping database node, so the database node allocation may be viewed as explicit rather than implicit. The L4 specification, however, does not *require* such an implementation.

  - In effect, EROS nodes reify capability storage, and require us to reason about kernel memory type safety in layered fashion. While the atomicity properties of the kernel interface make this possible, the constraints are difficult to understand and to reason about (formally or informally), and they significantly complicate the kernel implementation by introducing what may be thought of as cache coherency constraints across different representation caches.
  - EROS nodes do not provide a convenient representation of address spaces. Nodes have 32 slots, and this induces a structural constraint that shared subspaces must be expressed as aggregations of 32^k page units. In practice, this constraint has proven onerous for applications.

  The node structure is replaced in Coyotos by first-class kernel process structures and a new memory mapping structure called a *prefixed address translation tree*.

- EROS and KeyKOS intentionally omitted non-blocking messaging from the system primitive set. Similar effects can be achieved by using additional threads as message posting agents. Unfortunately, the result is both slow and pragmatically complicated. Without non-blocking notify, certain commonly used mutual exclusion patterns involving transmissions combining data and capability payloads through shared memory are very difficult to construct efficiently.

  Coyotos incorporates a non-blocking event posting mechanism.

- The current EROS IPC mechanism does not handle multithreaded receivers gracefully, and therefore fails to adequately encapsulate details of server implementation.

  Coyotos will incorporate explicitly named communication endpoints.

- EROS implemented (in the kernel) per-process capability registers. This proved to be constraining for applications that needed to manage large numbers of capabilities. A capability address space model was introduced late in the EROS design cycle, but was never integrated effectively into the capability invocation operation.

  Coyotos will provide more direct support for capability address spaces.

Explicit communication endpoints are a new feature in Coyotos, but with this exception all of the differences mentioned above are simplifications of the existing system that preserve all of the existing EROS design properties and constraints. The revised invocation mechanism can likewise be implemented without violating the EROS design constraints. While there are "systems" experiments that we intend to conduct with Coyotos, we are explicitly trying to restrict the core Coyotos architecture to refinement and simplification rather than invention.

By far the most significant change in the Coyotos effort is that the kernel implementation, and the re-implementation of critical system services borrowed from

EROS, will proceed using a systems programming language with a mechanically specified formal semantics.

BitC: A Language for Systems Programmers

We have identified in the introduction the main deficiencies of existing languages from the perspective of kernel development: the absence of machine-level representation types and data layout control, and the inability to write programs that run without dynamic allocation. Clearly, we require a language with an unambiguous formal semantics that can be mechanically captured.

It is often stated that aliasing is a fundamental impediment to analysis in languages such as C. While true, we suggest that this view is misleading. C introduces many unnecessary aliasing concerns, but the problem of aliasing cannot be eliminated by subsetting the C language. Kernels are inherently alias-intensive programs, and reasoning about the effects of assignments through aliases is an unavoidable part of the problem of kernel verification. For this reason, we have *not* listed alias elimination as a language requirement. Pragmatically, it is extremely helpful to have a language in which idiomatic "false" aliasing can be eliminated or reduced. It is also helpful to have a language in which the use of idiomatically induced assignment can be eliminated, e.g. through use of tail recursion as an alternative to looping constructs.

One advantage to writing a workshop paper *after* the workshop is that the paper has the opportunity to reflect some of what has been learned in the workshop. In our case, the impact has been substantial. At the workshop, we introduced BitC as a language in the intersection between Scheme and C. We added machine-level representation types and C-style structures to Scheme, eliminated operations that allocated storage (including closure values), and prohibited mutation of local variables. One goal of the BitC design was to arrive at a language that could be directly emitted to C in a *very* small code generator — small enough to be credibly validated by inspection. Eventually, we intend to generate machine code directly.

Following the discussions at the workshop, our ideas about BitC evolved significantly. We came to realize that significant transformations on BitC programs would be required to rewrite programs into a form suitable for direct code generation, and that these transformations would ultimately need to be verified. While the kernel subset language must still avoid dynamic allocation, the key issue in the language from a verification perspective is termination reasoning rather than dynamic allocation. This led us to reframe some of our restrictions:

- BitC must enable the developer to straightforwardly author programs that do not dynamically allocate storage (and we need to provide tool support to check this). BitC need not *prohibit* dynamic storage allocation.
- The complete elimination of closure values was excessive. The actual requirement for kernel programs is to prohibit *upward escaping* closure values that

capture local bindings (because these require dynamic allocation). There is no difficuty in using closure values defined at top level, or closure values that might be hoisted to top level without alteration of meaning. This alteration allows us to re-introduce some idioms for data structure traversal that must otherwise be expressed less directly.

Subsequent to the workshop, we introduced several new design elements into BitC:

- Parametric polymorphism supported by pattern matching and a type inference mechanism.
- Tail recursion, but limited to the case in which all of the procedures participating in the tail recursion requirement are bound simultaneously in the same LETREC-like form.
- Higher-order procedures, but using a surface syntax that discourages curried invocations. Because the use of escaping closure values is prohibited in the kernel subset, curried procedures cannot be used within the kernel.
- An ML-like tuple and datatype model.
- Vector types

Finally, there was one obvious issue that we addressed only obliquely (as "C-style structures") at the workshop: the need for unboxed aggregates *and unboxed references*. These have now been incorporated into BitC. These features in turn require us to ensure that the temporal scope of references must be bounded by the temporal scope of the referenced object, but this appears to be straightforward.

The provisional result appears to be a language with an unambiguous formal semantics that can be used to implement critical applications (including the BitC compiler). BitC has a clean subset in which the kernel can be implemented. The resulting language should probably no longer be thought of as "an intersection of C and Scheme." Rather, BitC is best viewed as ML with representation types and unboxing, the module system excised, and a parsable, LISP-like concrete syntax. Also, BitC discourages currying in favor of tuplization. As the language design progressed, we gained new appreciation for the "minimal mechanism" character of ML. The foundational semantics of the current BitC language is not substantially larger that than of the workshop version. Matters are now sufficiently far along that we are examining how to introduce measure conjectures and theorem statements into the language, and considering how to mechanically capture the reasoning about termination of an `eval()` procedure when it is applied to a known-terminating program in a language that does not intrinsically impose total types.

High-Level Objectives

Creating a new operating system using a new programming language involves an absurd amount of work. It seems only reasonable to ask: what are we trying to achieve? Before answering this question, it is useful to describe the current context of high-assurance systems.

Our group was initially drawn to verification as a means of increased security assurance. We are dissatisfied with the level of confidence achievable under currently standardized assurance schemes, and would like to establish a stronger foundation for robust and secure systems.

Role of the *Common Criteria*

The most widely accepted statement of security assurance criteria today is the *Common Criteria* [6]. The highest assurance evaluation (therefore highest confidence) level in the *Common Criteria* scheme is known as EAL7. Its requirements may be summarized as:

- Rigorously state your threat model and functional requirements.
- Formally state your security policy and a model of the system. Rigorously state how the security policy addresses your threat model, and how the system model addresses your functional requirements.
- Verify formally that the policy is enforced in the model of the system
- Show rigorously (as opposed to formally) that the implementation corresponds to the model.

The approach is basically sound. The last step can (and should) be strengthened to require formally verified correspondence. The decision to settle for rigorous correspondence demonstration was a pragmatic compromise reflecting the perceived state of the art in program verification circa 1980.

Our group has spent a fair bit of time laying the groundwork for this type of evaluation for EROS. As our understanding of the process has increased, and we have reached several conclusions:

1. The *Common Criteria* process is exceedingly difficult, not because it is conceptually hard to do but because it imposes an overwhelming burden of paperwork. The majority of this paperwork can be eliminated if formal methods are used where merely rigorous methods are currently required.
2. The process as currently defined has limited real-world value. At the end of the day, the customer isn't running the formal system model. They are running the code. Long experience shows that human inspection of code — and we believe this applies to *rigorous* inspection as well — is simply an inadequate source of practical security.

One solution to this is to extend the use of formal methods all the way to the code.

3. There are serious problems in the *Common Criteria* scheme and also in the evaluation process:
   - A few evaluation requirements of the scheme induce functional requirements that *reduce* the security of the final system.
   - No evaluation guidelines for evaluation above the EAL4 assurance level (soon: EAL5) exist.[6] In consequence, no public confidence is possible because it isn't understood what higher levels of assurance evaluation *mean*.
4. In the absence of widely and publicly deployed systems that have undergone a well-defined high-assurance evaluation process and been demonstrated by practical experience to be defensible, there exists no empirical evidence that the *Common Criteria* process works at all. In light of this, the cost of high assurance evaluation is not objectively justified.

   There is clear evidence from other domains, notably FAA Level-A flight control systems, that the use of full formal methods is an effective means of achieving robustness. It is likely, but not known, that this success extends to situations where proactive attempts at compromise enter the picture, *provided* that the threat model and requirements have been adequately captured. Unfortunately, no technique is known that ensures exhaustive threat or requirement modeling.
5. The *Common Criteria* process embodies a fundamental and irreconcilable conflict of interest: the party who creates the software has fiduciary influence over the party who evaluates the software, and the absence of transparency in the process lends itself to misuse and even abuse.

   Evaluation customers (the software providers) form a "buyers cartel." The absence of a large supply of business for evaluators creates an economic environment in which the diligence of the evaluation itself becomes negotiable. Evaluators are understandably reluctant to confirm this publicly, but it is widely acknowledged privately as a pervasive problem. The quality standards of the U.S. certified evaluation providers have been steadily deteriorating since the *Common Criteria* process was first deployed.
6. Taking these issues together, the *Common Criteria* serves primarily as a means of protecting incumbent vendors to governments rather than a tool for improving objectively measurable security.

As a result of these issues, Shapiro recommended in response to inquiries from members of the United States Senate during the Clinton administration that the U.S. government "evaluated product" purchasing requirement be dropped

[6] As calibration, EAL4 is the current evaluation assurance level of Microsoft's Windows XP (and many other products). No EAL4 system can be reliably deployed in hostile environments (such as open networks).

for all but the most sensitive applications, and that the latter insist on and fund the mechanisms to produce EAL6 or better evaluation processes. The first recommendation appears to have been accepted. The second was not.

An Alternative

Taken as a methodology, there is much in the *Common Criteria* that is worth borrowing. We propose to overcome some of its weaknesses by extending the concept of open source systems to open proof. By "open proof," we mean systems in which:

- Source code for the software artifact is publicly accessible.
- A public statement of the requirements met by the system exists in both definitive formal specification and non-normative informal language.
- A full formal verification that the implementation meets these requirements has been performed using a publicly available proof engine.
- The resulting proof trail, sufficient to allow a third party to independently re-execute the verification, is published in machine-readable form.

The last point bears emphasis. In an environment where software must be adapted and customized by the customer, proof checking is insufficient. The customer must be able to re-execute the entire proof process on locally modified versions of the system.

Realistically, we do not expect that software customers will re-execute these proofs, nor that they would understand directly what the proofs mean. We *do* expect that customers facing potential liability in critical deployments may hire domain experts to check the results as part of software acceptance qualification.

Ultimately, our objective is to redefine the standards of acceptable practice in critical software by demonstrating publicly that formal methods are *not* "too hard" or somehow impractical. If we succeed, the "trust me" approach to software security will become economically non-viable.

Verification Goals for Coyotos

The properties that we would like to verify for Coyotos can be divided into low-level (tactical) properties about the implementation and overall system model correspondence properties.

Implementation Properties

Design Rules The Coyotos kernel inherits a (relatively short) list of design rules that help to reinforce both the atomicity and correctness objectives of

the kernel. Among these, the most important is the "two phase" rule. We would like to formalize and rigorously check this rule and several others that devolve from it. A few of these have recently been validated using control flow model checking [3].

Access Check Enforcement We would like to formalize the access rules for each type of system object, and verify that the actual implementation honors the capability-defined access rights at all appropriate points.

Semantic Observability A difficult check we would like to validate is to formalize what is meant by "semantic observability" and verify that the prepare phase does not make semantically observable modifications to the system state.

The Constructor Assumptions The constructor verification relies on the assumption that certain kernel-implemented capabilities were trivially safe and that weak capabilities are transitively read only. Both of these assumptions should be verified.

Address Translation Because address translation data structures might violate both the type safe heap of the kernel and the overall security of the machine, we wish to verify that the algorithm by which the hardware memory map is constructed implements a correctness-preserving translation from the software-defined mapping structures.

Serializability In the SMP version of the kernel, we would like to verify at the end of each kernel invocation that there exists some sequential, non-overlapping sequence of kernel calls that can account for the system state.

Memory Safety We would like a kernel that is known to be "mostly memory safe." Certain hardware data structures, most notably the process and memory management structures, necessarily require low-level manipulation.

Space Bank Isolation Contract The Coyotos storage allocator must ensure that no resource is simultaneously allocated to more than one requestor. This so-called "exclusively held" property is foundational for confinement and higher level mandatory policy. It should be possible to formalize and verify this property.

System Model Correspondence Properties

Our earlier work on confinement verification yielded a formal system model in which the system state and the key system operations were formalized in an operational semantics for an abstracted machine. As a practical matter, this formalization was too high level to be useful for correspondence checking of the real kernel implementation.

We would like to create a more detailed abstract system model, and show that there are fairly direct correlations between this abstract system model and our actual implementation. What this means in practical terms is something we are reluctant to speculate on until we understand this part of the process better.

Conclusion

Last year, Shapiro authored a controversial column for IEEE Software entitled *Understanding the Windows EAL4 Evaluation*. While the column was widely (and correctly) taken as an indictment of the Microsoft evaluation, astute readers recognized in it a much deeper indictment of the *Common Criteria* process and the current state of computer security in the wild. The current state of affairs in both security and reliability is unsustainable.

Progressive adoption of software verification techniques in critical systems offers the possibility of a major improvement in the robustness and security of day-to-day systems. Our hope with the Coyotos project is to demonstrate that these methods are much more realistic today than is widely understood. We intend to craft a set of tools for creating more robust, high-efficiency system code and provide a publicly accessible, well-documented exemplar for how the tool is applied. Along the way, we intend to create an operating system platform that might be suitable for use in critical applications including critical infrastructure, life-critical systems, and operationally critical business applications.

References

1. Bartlett, J. F: Scheme!C: A Portable Scheme-to-C Compiler. Technical Report WRL Research Report 89/1, Digital Western Research Laboratory, Jan 1989.
2. Bevier, W. R.: Kit: A Study in Operating System Verification. IEEE Transactions on Software Engineering. **15**(11). 1989. pp. 1382–1396.
3. Chen, H., Shapiro, J. S.: Using Build-Integrated Static Checking to Preserve Correctness Invariants. Proc. 2004 ACM Symposium on Computer and Communications Security. Oct. 2004.
4. Hardy, N.: The KeyKOS Architecture. Operating Systems Review **4**(19), Oct. 1985, pp. 8–25.
5. Guttman, J. D., Ramsdell, J. D., Swarup, V.: The VLISP Verified Scheme System. Lisp and Symbolic Computation, **8**(1-2), 1995, pp. 33–110.
6. —: Common Criteria for Information Technology Security, International Standards Organization. International Standard ISO/IS 15408, Final Committee Draft, version 2.0, 1998
7. Kaufmann, M., Moore, J. S.: Computer Aided Reasoning: An Approach, Kluwer Academic Publishers, 2000.
8. Lampson, B. W.: A Note on the Confinement Problem. Comm. ACM. **16**(10), 1973, pp. 613–615.
9. Neumann, P. G., Boyer, R. S., Feiertag, R. J., Levitt, K. N., Robinson, L.: A Provably Secure Operating System: The System, Its Applications, and Proofs. Computer Science Laboratory Technical Report CSL-116, 2nd ed., May 1980, SRI International.
10. Rajunas, S. A.: The KeyKOS/KeySAFE System Design Tehnical Report SEC009-01, Key Logic, Inc., March 1989.
11. —: L4 *eXperimental* Kernel Reference Manual. System Architecture Group, Dept. of Computer Science, Universität Karlsruhe. 2004

12. Sinha, A., Sarat, S, Shapiro, J. S.: Network Subsystems Reloaded. Proc. 2004 USENIX Annual Technical Conference. Dec. 2004
13. Shapiro, J. S., Smith, J. M., Farber, D. J.: EROS, A Fast Capability System. Proc. 17th ACM Symposium on Operating Systems Principles. Dec 1999, pp. 170–185. Kiawah Island Resort, SC, USA.
14. Shapiro, J. S., Weber, S.: Verifying the EROS Confinement Mechanism. Proc. 2000 IEEE Symposium on Security and Privacy. May 2000. pp. 166–176. Oakland, CA, USA
15. Shapiro, J., Vanderburgh, J. Northup, E, Chizmadia, D: Design of the EROS Trusted Window System. Proc. 13th USENIX Security Symposium. 2004
16. Tarditi, D., Lee, P., Acharya, A.: No Assembly Required: Compiling Standard ML to C. Letters on Programming Languages and Systems. June 1992.