# Using Build-Integrated Static Checking to Preserve Correctness Invariants*

Hao Chen
University of California, Berkeley
hchen@cs.berkeley.edu

Jonathan S. Shapiro
Systems Research Laboratory
Johns Hopkins University
shap@cs.jhu.edu

## ABSTRACT

A key missing link in the creation of secure and robust systems is finding a cost effective way to demonstrate and preserve correspondence between a software design and its implementation. This paper explores the use of software model checking techniques to validate selected design invariants in the EROS operating system kernel. Several global consistency policies in the EROS kernel can be expressed as finite state automata. Using the MOPS static checker, we have been able to validate the EROS kernel implementation against these automata. In the process, we have confirmed the practical utility of the basic verification technique, identified a number of desirable enhancements in MOPS, and located bugs in the EROS implementation.

A key contribution of this paper is establishing that it is practical to integrate software model checking into normal development life cycle. Model checking is efficient enough that it does not add noticeably to our build times. This allows us to view it as a tool for error *prevention* rather than *detection*. Our work with EROS and MOPS suggests that domain specific application of software model checking is a practical and powerful technique for software assurance and maintenance.

## Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*verification*; D.2.4 [**Software Engineering**]: Software/ Program Verification—*formal methods, model checking*

## General Terms

Security, languages, verification

## Keywords

Security, model checking, verification, static analysis, assurance, MOPS, EROS

*This research was supported in part by NSF CCR-0326577.


*CCS'04,* October 25-29, 2004, Washington, DC, USA.


## 1. INTRODUCTION

Static checking tools have seen broad success in the last few years. Tools like BLAST [1] and MECA [2] have been applied to various open source projects – most notably Linux – with considerable success. Engler reported discovering 7 confirmed race condition errors in the Linux 2.5.62 kernel using RacerX [3]. Yang *et al.* report 36 confirmed user pointer bugs in a later version of Linux[1] [2]. Zhang *et al.* report 4 authorization check errors within the file system code using CQUAL [4].

While it is pleasing to know that 47 Linux kernel bugs have been discovered, the practical impact of these efforts must be viewed skeptically. With the exception of SLAM [5], none of these tools have been successfully integrated into the software development life cycles of the systems they have checked. Given this, it is unclear whether developers can successfully create and maintain specifications in the course of normal development. Some of these tools (MC/MECA) remain unreleased. Given this, developer investment in either tool is risky, as it is questionable whether long-term support for them can be sustained.[2]

In this paper, we report our experiences in integrating the MOPS static checking tool [6] into the development cycle of the EROS operating system. MOPS is an open source, flow-sensitive model checker for temporal safety properties. It operates on unannotated C code using externally provided property specifications. MOPS has been used successfully to find security related bugs in mature, widely deployed open source programs [7]. EROS [8] is a relatively robust, mature, capability-based operating system. It has supporting formal access and system models [9], and is architecturally derived from an earlier system (KeyKOS [10]) with an exceptional mean time between failures (18 years).

Both the EROS design and its existing development practices incorporate strong measures to defend against developer error, and (ignoring active development code) the project group has discovered fewer than two bugs a year over the last eight years. In consequence, we did not expect to find

[1]We believe from examination of the paper that this was a late 2.5 kernel version, but the version number was not reported in [2].

[2]MC, which is derived from GCC and therefore "open source," has been reporting results for five years; the absence of a public release enabling independent evaluation should be cause for skepticism in the research community. BLAST is built in a programming language (ML) that lacks broad developer support and relies on two theorem provers (Simplify and Foci) that are not readily available for general use or third-party maintenance.

any significant number of errors using static checking. Our goal was to provide supporting infrastructure that could be integrated into the development process to prevent the *introduction* of certain errors into the code.

Because we are concerned here with development integration, our focus in this paper differs from earlier experiences reports:

- We are concerned here with bug *prevention*, rather than bug *discovery*. Experience with historical EROS development suggested several kernel invariants that were easy to get wrong. A key goal of the work reported here was to build automated checks of these properties to ensure that new errors did not occur.

- Because of the structure of the EROS system, the implementation *should* be "friendly" to static checking approaches. An alleged advantage of the MOPS approach is the ease of construction of specifications. We wanted to validate this by having someone who was not an author of the checking tool create specifications and check them.

- Our concern with scalability is driven by "time to check" rather than "lines of code." We wanted to get MOPS to the point where the checking process could be integrated into our compile and build process.

Robust software systems are difficult to design and maintain. Lightweight program analysis tools offer one promising approach for maintenance, but many of them are not suitable for use in operating system kernels. Dynamic (runtime) tools are inappropriate, because they impose significant overhead during production execution and do not ensure exhaustive checking (only the paths that are actually executed are examined). Static (compile-time) tools offer much better coverage of program paths. However, some static tools are designed to identify particular properties that were envisioned by the tool developer [11, 3]. Some general tools supporting user-specified properties, such as MC [12], are not "sound" – the failure to detect an error in such tools does not guarantee the absence of errors.

An orthogonal issue is that the checking technique must be modestly invasive of the source base. To achieve robustness, developers are prepared to revise both their checking tools and their programming idioms, subject to the requirement that these changes must not result in less comprehensible or maintainable code. Experience with annotation-driven checking tools [11] on a related project makes us skeptical of how "light" these techniques actually are. The number of annotations needed in one carefully written system quickly became explosive, and the resulting code was essentially unmaintainable.

To overcome these problems, we explored applying application specific software model checking as a middle ground approach for software assurance. We view the system as a collection of interacting state machines that collectively ensure end to end behavior. The transitions of these state machines can be expressed as temporal safety properties that can be exhaustively checked using a sound model checker. Rather than check all the properties of the target system simultaneously, we check the properties individually using the MOPS static checking tool. In the course of our exploration, we have gained confidence in both the target implementation and its design, discovered a small number of bugs, and identified a variety of ways in which the expressive power of MOPS could be improved without altering the simplicity of the basic checking technique.

From an engineering perspective, one key feature of the MOPS approach is that the specifications (finite state machines) can be written by ordinary programmers and the results (traces) are readily comprehensible. A second is that the MOPS execution times are short enough to let us integrate these checks into our regular build process.

## 2. OVERVIEW OF MOPS

MOPS is a static (compile-time) analysis tool [6] that checks if a program violates temporal safety properties. Temporal safety properties specify the requirement that programs perform certain operations in defined sequences, and can express many application security properties. The MOPS user describes a safety property in a finite state automaton (FSA). The FSA transitions on syntactic expressions represented by abstract syntax trees (AST). Any sequence of transitions that ends in the final state of the FSA is deemed to violate the safety property.

MOPS consists of a parser and a model checker. The parser compiles a C source program into a control flow graph (CFG). Then, the model checker decides if any path in the CFG may violate the property represented by an FSA. Algorithmically, the model checker creates a Pushdown Automaton (PDA) to represent all the paths in the CFG, intersects the PDA with the FSA, and decides whether the intersection (also PDA) is empty. If not, MOPS reports error traces, which are program paths that violate the property, to help the user identify program errors.

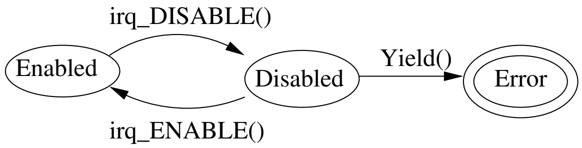
For example, MOPS might be used to check the Yield Property: the program should not call `Yield()` when interrupts are disabled. (Section 4.1 explains this property in details). The kernel may enable an interrupt by calling `irq_ENABLE()` and disable an interrupt by `irq_DISABLE()`. The MOPS user describes this property by an FSA. Figure 1(a) shows this FSA, and Figure 1(b) shows a code fragment that violates this property.

In large software systems, invariants such as the Yield Property are hard to maintain because (a) they do not tend to respect modularity boundaries, and therefore cannot be maintained in a single, well-localized place in the source code, and (b) many such invariants are in effect simultaneously. The programmer is forced to maintain a mental model of global program behavior, and attempts to maintain the invariants against the model while making localized changes to software. Too often this process fails, either because the necessary mental model is too complex or the number of simultaneous invariants exceeds the programmer's simultaneous reasoning capacity.

### 2.1 Soundness

MOPS strives for soundness and scalability. A sound tool will not overlook any violations of the safety property in the program[3], and a scalable tool can check large programs with moderate computational resources. In the current state of the art, however, perfect soundness results in poor scalability. Since MOPS is designed to be a practical tool for checking safety properties on large programs, it tries to strike a

[3]In other words, the tool is sound for the theorem "the program satisfies the safety property"




(a) An FSA describing this property.

```
f() {// Interrupts are enabled
  if (condition)
    g();
  Yield();
}
g() {
  irq_DISABLE();
}
```

(b) A program fragment that violates this property. One path through the function `f()` satisfies the property, but the other path violates it.

**Figure 1: An FSA describing the property that "*the program should not call `Yield()` when interrupts are disabled*" and a program fragment that violates it.**

balance between soundness and scalability. To strive for soundness, MOPS is path sensitive, which means that it follows every path in the program (including arbitrary number of iterations of loops) except a few minor cases discussed below.[4] MOPS is also context sensitive, which means that it can match each function return with its call site. Since data-flow analysis presents many difficulties for scalability, MOPS chooses to be data-flow insensitive. In general, MOPS ignores most data values in the program and assumes that each variable may take any value. Because MOPS is value-insensitive, it assumes that both branches of a conditional statement may be taken and that a loop may execute anywhere from zero to infinite iterations. As such, MOPS is mostly suitable for properties that are control-flow centric.

MOPS is sound under the following intrinsic assumptions:

- The program is single-threaded.
- The program is memory safe, which implies that the program has no buffer overrun bugs.
- The program is written in standard compliant C, augmented by selected GNU C extensions. For example, MOPS chooses some particular resolution of control flow order where the C specification is ambiguous: programs that rely on the ordering of control flow in such computations are not standard compliant C programs. As another example, MOPS does not understand inline assembly code, but it can treat assembly functions as "external" procedures.
- The program does not violate the soundness assumptions required by the user-specified temporal safety property. Some properties are sound only under certain assumptions of the program. For example, to enable the user to express the property "do not call `open()` after calling `stat()` on the same file name", MOPS allows the user to declare a generic pattern variable $x$ and then use `open(x)` and `stat(x)` in the FSA. In this case, the variable $x$ in `stat(x)` and `open(x)` refers to any variable that is syntactically used in both `stat()` and `open()`. Since pattern values are based on syntactic matching, they do not consider value flow or liveness violations: if the program has "`stat(f); g=f; open(g)`", then MOPS does not know that $g$ is an alias of $f$. Similar problems can arise if the value of $f$ is modified by means of an aliased reference between the `stat(f)` and `open(g)`.

The current implementation of MOPS does not consider control flows that are invisible in the control flow graph, such as indirect calls via function pointer, signal handlers, long jumps (`setjmp()`/`longjmp()`), and libraries loaded at runtime (`dlopen()`). As it happens, the EROS kernel does not use `longjmp()`.[5]

## 3. OVERVIEW OF EROS

EROS is a capability-based operating system that runs on commodity hardware. Applications invoke user-implemented objects and kernel services by invoking kernel-protected capabilities. Our focus of attention in this paper is the EROS microkernel, and specifically the invariants of the microkernel. A more complete discussion of the EROS design can be found in [8, 13].

One view of a microkernel is that it implements an extended virtual machine that is derived from and built upon the underlying hardware architecture. In microkernels, this extended virtualization generally provides a primitive interval timer, multitasking, virtual memory, and exception encapsulation. In addition, microkernels provide a small set of system calls necessary to direct and manipulate the virtualization performed – to define memory maps, set scheduling policy, and so forth. Allowing for some variations in individual implementations, there are either three or four substantive entry points in a typical microkernel:

1. The interval timer interrupt, whose arrival conditionally results in a scheduling preemption.
2. The page fault interrupt, which signals to the microkernel that it either needs to translate virtualized mappings into hardware mappings or request application-level action.
3. The system call trap, which directs the performance of a microkernel service, possibly an interprocess operation.
4. The major preemption entry point, which is invoked when the current kernel thread of control has become

[4] Although due to the lack of data flow analysis, MOPS may check paths that are infeasible in running programs and may report errors on these infeasible paths, it is still a sound path-sensitive analysis tool; since it never merges the analysis of conditional branches, it will not overlook property violations that a flow-sensitive but path-insensitive tool may miss.

[5] The kernel debugger uses `setjmp()` to establish a recovery point, but this code is not compiled in to the production kernel.

blocked and a new thread of control must be selected. This entry point only exists in interrupt-style kernels, including EROS.

While other hardware exceptions have associated entry points, microkernels generally implement these as "fast path" entry points whose effect is to record fault information in the process control block of the currently executing process and then invoke the major preemption entry point. We will not consider exceptions of this kind further in this paper.

Three aspects of the EROS kernel design are particularly relevant to model checking: the fact that it is an interrupt style kernel, the single level store, and the state caching design of the system.

**Interrupt-Style Kernel** An interrupt-style kernel [14] is one in which processes do not retain a kernel stack while blocked. On wakeup, the process resumes by restarting the system call that originally caused it to sleep.

This style of kernel design imposes a transaction-like structure on the execution of system calls. Kernel service invocations proceed by testing preconditions, reaching a "commit point," and then performing the requested operation. No externally visible modifications are permitted prior to the commit point, and after the commit point the service invocation is required to run to completion successfully. If a precondition cannot be satisfied, the requesting process is placed on a queue and is required to restart its invocation from scratch. One way to think of this is that every system service invocation carries an implicit "catch block," and that any cause of blocking involves a "throw" that is caught at the system call boundary.

The transactional style of execution naturally results in temporal safety invariants that must be observed. Procedures that cause externally observable state changes must not be called prior to the commit point. Procedures that might remove objects from memory must not be called after the commit point has been reached. Procedures that check preconditions (e.g. Is the object we are mutating writable?) may be viewed as causing typestate transitions [15, 16] in a temporal safety property. Many of the key temporal safety invariants of interrupt-style kernels can be expressed as finite state machines.

**Single-Level Store** The EROS system implements a transparent single-level store built on a global snapshot and checkpointing system. One consequence of this design is that an object descriptor (a capability) refers to an object that can be either in memory or on disk. Descriptor usage can conditionally induce object faults, and transient pinning is used to ensure that objects required in a particular operation remain in memory for the duration of the operation. This induces restrictions on what procedures can be permissibly called at what points in the kernel control flow. After a commit point is reached, no operation is permitted that might result in an object removal. Prior to a commit point, no externally visible modifications to kernel state are permitted.

**Caching Design** EROS implements a caching approach to managing system state [17]. Process and memory map state is stored in user-allocated, kernel-protected data structures called *nodes*. Hardware-manipulable page tables and process control blocks are managed as a software-managed write-back cache of the state in these nodes [13]. This software-management strategy demands data structures for dependency tracking. EROS requires that the code that maintains these data structures must follow certain control flow invariants, many of which can be checked by a model checker. For example, one invariant requires that the code should not modify any page table entry without a preceding call to create an entry in the data structures for dependency tracking.

Collectively, the state machines associated with these properties govern the majority of the EROS kernel's function. If we can establish through formal techniques that the transition rules of these state machines are satisfied, considerable confidence emerges in the global structure of the system. Better still, these formal checks serve as a guard against inadvertent error during maintenance. Model checking will not allow us to find local errors in the code, but may significantly reduce exposure to violations of global design rules. The aggregate constraint behavior of a large, complex system results from complex interactions between relatively simple individual state machines that guard particular aspects of overall consistency. By model checking these state machines individually, with some attention to the interaction points between them, the satisfaction of complex overall constraints can be assured.

# 4. PROPERTIES AND EXPERIMENTS

To evaluate the effectiveness of MOPS in establishing assurance, we modeled five properties of the EROS implementation. The first three can be expressed purely as control flow invariants, and fall directly within the space of properties that MOPS was designed to check. The remaining properties are typestate properties [15, 16]. We were interested in part to determine whether we could adequately approximate these using MOPS pattern variables, and also to determine whether it would be worthwhile to extend MOPS or some related tool with support for typestate.

It should be emphasized that EROS is in many respects an ideal subject system for this sort of analysis, and in consequence a frustrating one. The history of "design by invariant" in EROS and its predecessor KeyKOS collectively extends back over 30 years. This has the effect of making the system friendly to the type of analysis reported here. In the eyes of the checker, however, it is also somewhat frustrating: invariant discipline is very effective in resisting bugs, and there were relatively few to find. To safeguard MOPS against implementation errors that cause MOPS to miss bugs, for each property we artificially inserted bugs into the EROS kernel and ensured that MOPS found them.

## 4.1 Interrupt Enable and Disable, Yield

The simplest property we attempted was ensuring that interrupt enables and disables are properly bracketed. Every disable should be balanced by a corresponding enable, and there should not be redundant enables. In addition, the `Yield()` function (which abandons the current system call) should not be called while interrupts are disabled. This is because `Yield()` abandons the current kernel control flow and invokes a kernel system call, but all kernel system calls are expected to be invoked with interrupts enabled. The FSA in Figure 2 describes this property.

A problem with checking interrupt enable and disable is that it requires a stack or a counter. Finite state automata are not powerful enough to implement counters, but in practice the enable/disable depth of a well-structured kernel is not deep. To cover the current behavior of the kernel, we

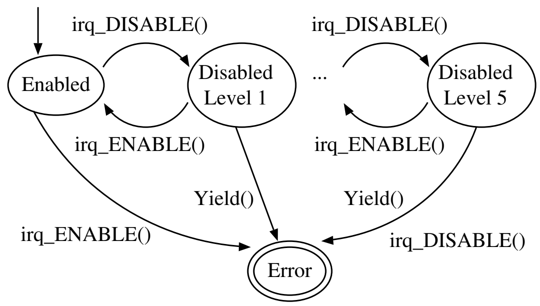



**Figure 2: An FSA describing the property:** ***(1) each*** ***`irq_DISALBE()` should be balanced by an `irq_ENABLE()`; (2) there should be no redundant `irq_ENABLE()`; (3) `Yield()` should not be called while interrupts are disabled.***

implemented five distinct disabled states, and used these to simulate up to five levels of nesting. Additionally, we used the error state as a guard state to validate that five levels were enough. The need for fixing a maximum number of nesting levels came as an unpleasant surprise; one of the ancillary outcomes of this analysis was a decision to review the use or interrupt nesting to reduce this depth.

Consistent with our expectations, we did not find any violations of this property. We *did* modify the EROS source code to convert a table-driven capability handler dispatch mechanism into an equivalent switch-driven mechanism so that the checker would be able to see that these routines are reachable. The resulting code is functionally equivalent and probably more efficient, as the compiler now has the option to implement the dispatch using a vector of branch instructions. We also modified one place to rewrite a correctly conditionalized disable/enable pair so that the static checker would not report a false positive error trace. This was a false positive that could readily be resolved through conditional liveness analysis [16].

## 4.2 Yield, Commit

As has previously been mentioned, EROS is an interrupt-style kernel. The flow of operation proceeds by first checking preconditions and then performing the operation itself. These two phases are separated by what we call a "commit point." A kernel invocation can yield before the commit point, but not after. This constraint leads to two rules:

- Every system call control path should invoke exactly one of `Yield()` or `Commit()`.
- Following a call to `Commit()`, it is a bug to subsequently call `Yield()`.

The `Yield()` function does not return, so we do not check for `Commit()` after `Yield()` or `Yield()` after `Yield()`. The FSA in Figure 3 describes these rules.

The "yield or commit" design rule is fundamental to the EROS security model. The consistency of our security model is predicated on an argument about atomic, stepwise correct evolution of the protection graph [9]. The yield or commit design rule is the rule that guarantees the atomicity of the protection graph transformations. This was in fact the property that we attempted first, because we were struck by how

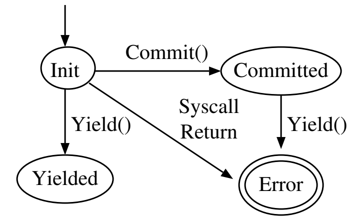



**Figure 3: An FSA describe the property:** ***(1) every system call control path should invoke exactly one of `Yield()` or `Commit()`; (2) following `Commit()`, the kernel should not call `Yield()`.***

direct this property corresponds to the previous formal verification work on the EROS confinement mechanism [9].

Sadly or happily (depending on which author you ask), we found a bug in the course of this analysis. After a commit point, when the EROS kernel is midway through a transition to more dynamic heap management, one system service call used by user-mode drivers allocates heap memory from a finite kernel pool by calling `malloc()`. Should this allocation fail, `malloc()` calls `Yield()`. This violation of the "yield or commit" design rule revealed that we had entirely failed to consider this challenge when we decided to introduce `malloc()` into the kernel. The correct repair in the context of the current EROS design is to allocate all required memory during a "dry run" phase prior to the commit point. Fortunately, this is the *only* system service in the EROS kernel that invokes `malloc()` after startup initialization has completed, so we do not anticipate great difficulties in correcting this flaw.

## 4.3 Sleep and Yield

One very effective way to create problems in a kernel is to mishandle process sleep and wakeup. Kernels have a large number of stall queues, and mishandling of these queues can easily lead to lost processes or data corruption. Because stall queue manipulation occurs with interrupts disabled, these errors are difficult to debug. Sleep handling in EROS differs from widely-used kernels in two significant regards:

- A process can be asleep on at most one stall queue at a time. There is no equivalent to the UNIX `select()` operation in the EROS kernel.
- The `Sleep()` and `Yield()` operations are not atomically joined. In an interrupt-style kernel, it is sometimes convenient to place the process to sleep on the appropriate stall queue and to perform additional processing using the kernel thread of control before relinquishing the processor.

The first property implies that no path through the kernel should call `Sleep()` more than once, and the second implies that every call to `Sleep()` should be followed at some later point by a call to `Yield()`. The FSA in Figure 4 describes these properties.

There is a third property that is almost but not universally true: nearly every call to `Yield()` follows a `Sleep()` of some sort. Conceptually, this is because a `Yield()` generally means that there was some reason that an operation could not be completed immediately, and all of the operations that might eliminate such an impediment signal their

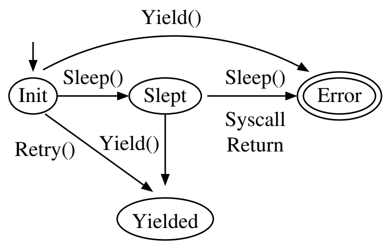



**Figure 4: An FSA describing the property:** ***(1) no path should call `Sleep()` more than once; (2) after a `Sleep()` call, the kernel must call `Yield()`; (3) before a `Yield()` call, the kernel must call `Sleep()`, unless it has called `Retry()`.***

completion by waking up a stall queue somewhere. Given this, one would expect that a yielding process should be on a stall queue somewhere, which is achieved by calling `Sleep()`.

When we originally specified this property as above, we neglected an important exceptional case. There are certain cases where the kernel *may* be in an inconsistent state but does not know for sure. For example, a system call or a page fault may rely on the continued existence of a page table entry, but may need to bring an object into memory, which may invoke the aging logic to evict something to make room. The eviction may invalidate one or more page table entries, and these page table entries may turn out to be the ones that the current operation is relying on.[6] It is not cost-effective to remember which page table entries are specifically implicated by the current call; instead, the kernel keeps a global boolean flag `PteZapped`, which is cleared on entry to the kernel and is set whenever a page table entry is invalidated. This approach is conservative, but effective.

In defensive cases such as the one described above, it is appropriate for the current operation to perform a `Yield()` voluntarily. Because the process has not gone to sleep, it has not relinquished its ownership of the CPU, and will retry the current operation. Assuming that there were no real impediments to completion, the system call will complete successfully during the second pass.

To capture both the property and the intent correctly, we modified the source code to call `Retry()` in these special cases (which is simply a wrapper for `Yield()`), and we modified the property to stop in a successful state whenever `Retry()` is called, as shown in Figure 4. With this modification, no further errors were found in the kernel.

## 4.4 Prepare Before GetRegs

The next property validates one of the caching requirements of the EROS kernel. Certain process-related operations, such as `proc_GetRegs32()`, operate by fetching values from the process control block. Because the process may not be active when registers are requested, it is necessary to encache the process first by calling `proc_Prepare()`. The property that we would like to check here is a typestate property: any call to `proc_GetRegs32(`*p*`)` requires that the typestate of process *p* is "cached." The effect of `proc_Prepare(p)` is to change the typestate of process *p* from "unknown" to

[6]We discovered this bug the hard way in 1998, and had a fun time working out how to build an effective torture test for it.

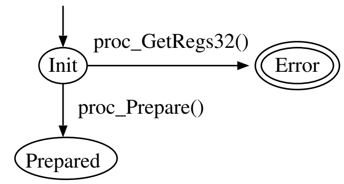



**Figure 5: An FSA describing the property:** ***before a `proc_GetRegs32()` call, the kernel must call `proc_Prepare()`.***

"cached."

MOPS does not implement typestate checking, but it *does* provide a feature called "pattern variables." Using pattern variables, we can statically verify that given a code sequence of the form

```
proc_Prepare(param1)
...
proc_GetRegs32(param2, &regset)
```

the arguments passed in as *param1* and *param2* are textually identical (by which we mean that their abstract syntax trees match). This is a much weaker check than the typestate check, but it is pragmatically sufficient for our purposes. The EROS kernel does, in fact, use the same variable in both calls, and we know from the implementation that any attempt to decache the process must ultimately call either `proc_NeedRevalidate()` or `proc_Unload()`. As long as neither of these functions is called, a cached process will remain cached. Therefore, so long as the value of *param1* has not changed before the call to `proc_GetRegs32`, we know that the process remains cached. The FSA in Figure 5 describes this property.

Note that because of the possibility of an intervening decache, the property we are trying to test is truly a typestate property; static types are insufficient to describe the control flow constraints we have described. The point of our trial was to understand how, in practice, typestate analysis would be helpful in validating properties about this sort of kernel. Our conclusion is that typestate is an essential tool for validation of this sort, but we note that in an interrupt kernel a surprising number of typestate properties can be checked without risk of ambiguities that might arise from failures of alias analysis. In this case, it is sufficient to know (a) that the argument process *p* was cached, (b) that this value of *p* reached the call to `proc_GetRegs()`, and that (c) there was no intervening decaching operation on *any* process.

We found no error in checking this property.

## 4.5 PTE Dependency Tracking

Our final check attempted to validate the consistency of the EROS memory management subsystem. EROS applications specify their address space structures using a tree of fixed-size capability lists (c-lists). The kernel traverses these structures on demand to construct page table entries. If a capability in one of the translated c-lists is subsequently overwritten, the derived page table entries must be invalidated. To ensure this, EROS maintains a dependency tracking data structure known as a *depend table* that maps from capability addresses (the address of the traversed slot in the c-list) to page table entry addresses. Whenever a c-list entry is overwritten, the depend table is used to invalidate the appropriate page table entries.

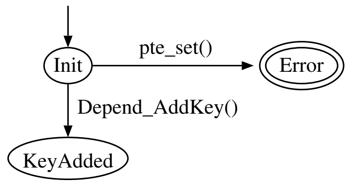



**Figure 6: An FSA describing the property: *before a `pte_set()` call, the kernel must call `Depend_AddKey()`.***

The design rule that we wanted to check is that no page table entry should be modified without a preceding call to create a depend table entry. This test is approximate. Since MOPS does not provide either typestate or value tracing, we did not attempt to check that the depend table entry was associated with the *right* page table entry. The overall goal of the algorithm is to ensure that every path in the hardware mapping table tree corresponds to some path in the node tree. The FSA in Figure 6 describes this design rule.

The check of this property revealed five places where page table entries were set without any dependency entry construction that MOPS could detect. Examination of the cases was sufficient to show that all five statements were correct, and that all five in fact *did* obey the property, but that this could not be detected statically. The cases arose because of an optimization in which a restarted page fault could bypass translation steps that are known to have been previously performed, because the required dependency table entries have been constructed in the previous pass. It is not entirely surprising that this check fails. The optimization in question is based on a correspondence between two trees of different arity in which the second tree (the mapping table hierarchy) is a lazily generated partial projection of the first (the node tree) [13]. Because of the difference in tree arity, the correctness of the correspondence – never mind the implementation – is not immediately obvious, and the correct implementation of this projection by the code is less so.

Matters are further complicated by the fact that the algorithm takes advantage of reverse correspondences between the two trees to avoid traversing the more expensive mapping structure when possible. The algorithm relies heavily on checks of values produced by previous passes through the translation code to determine the correct translation strategy. It is not clear whether the temporal safety properties of this code can be validated even with typestate support, because the automaton is unable to consider the cumulative effect of successive invocations of the address translation subsystem.

In our view, there are two lessons to take away from the failure of this check. The obvious one is that not every invariant can be reduced easily to temporal safety properties. The less obvious one is that *most* invariants can: of the many invariants that we attempted to check in the EROS kernel, this is the only one that failed entirely. Others would have required typestate analysis, but clearly fall within the scope of what model checking technique can do.

# 5. EVALUATION

We argue that usability and performance are two key criteria for evaluating the feasibility of integrating a static analysis tool into software development process. The usability criterion measures how easy it is for non-tool developers to write domain specific properties, and how disruptive the tool is to the source code and to the build process. The performance criterion measures how much overhead the static checking adds to the build process.

## 5.1 Usability

### 5.1.1 Property Specification

The five properties that we checked in Section 4 are design invariants in the EROS system. Historically, EROS has a long track record of specifying invariants before implementing them. While these specifications have not been collected in a single place (but ought to be), they have been published in the form of email discussions, design notes that are part of the source tree, and the original KeyKOS design document [18].

In the course of this experiment, our use of MOPS was iterative. It took us just a few minutes to write the initial property specification, but we would check this property and found that it produced errors because the specification was in some way incomplete. Then we would add new states and transitions to the specification to address the deficiency and would re-run the property. Two or three iterations of this were generally sufficient to capture the property accurately. We wrote the properties described here during the course of roughly sixteen hours spread over several conference trips; a long attention span was not required.

This experiment is a joint effort between the two authors: one being an expert only on the tool (MOPS) and the other an expert only on the system (EROS). Although they jointly formalized and checked some properties presented in this section, the system expert alone easily formalized and checked other properties, including several ones that are not discussed in this paper. This experience suggests that MOPS can be used successfully with minimal training on the tool by ordinary developers that know only the system well.

In general, we found few false traces with the refined FSAs. Most of these were due to failures of conditional liveness analysis, and were resolved by making minor modifications to the EROS code. In contrast to some other tools we have used, the modifications yield more readable and more maintainable code. In one or two cases we were unable to eliminate the last one or two false positives within a reasonable amount of effort. These cases and their explanations are now documented in the source code.

### 5.1.2 Integration into the Build Process

MOPS is minimally disruptive to the source code and to the build process. As we mentioned before, MOPS checks GCC-compliant C code and does not require code annotation. Running MOPS on EROS requires two steps: first, build a CFG for each source file and link all the CFGs together; second, model check the linked CFG against each property respectively. MOPS's parser produces CFG files in exactly the way that GCC's parser produces object files.

Integrating the MOPS processing into the EROS build process required only minor changes to the existing Makefiles. After these changes, the Makefiles automatically maintain the CFG files as part of the normal build cycle.

## 5.2 Performance

From the developer perspective, a key concern with MOPS was the run time cost of generating and building the CFG files. The EROS kernel has approximately 25,704 lines of code, of which 4,153 are header lines that are widely included. Building the default configuration of the EROS kernel without MOPS checking takes 12.13 seconds. When running the current version of MOPS on EROS, building and linking the same configuration takes 31.34 seconds, the majority of which is spent in Java I/O. Model checking all the five properties in Section 4 takes 100.13 seconds. This is a large enough addition of time that we tend to omit it during the "edit compile debug" loop, but small enough that we can make it a routine part of our major builds without noticeable cost.

Earlier versions of MOPS suffered from naively implemented I/O. The I/O code was easily improved, and the actual processing time used for managing the CFG was never a source of concern. However, the Java stream I/O package is difficult to use efficiently, and garbage collection is not well suited to applications that retain large graph structures for extended processing. Our sense is that an implementation crafted in a more conventional systems programming language might well yield a factor of ten improvement in the performance of the checking phase, and reduce the CFG construction times down to something less than the normal compile phase. This would be sufficient to let us run MOPS in every build we do.

## 6. DESIRABLE ENHANCEMENTS TO MOPS

The initial goal of MOPS was to check control-flow centric properties in security-critical application programs. When we tried to use MOPS on EROS, we found that we needed to improve MOPS in several minor ways that were described in Section 2. Several desirable improvements emerged that require non-trivial changes to the design of MOPS, and we have started work to implement these enhancements in future tools. We restrict ourselves here to improvements that preserve the desirable properties of MOPS: soundness, minimal invasiveness, developer-specifiable properties, and developer-comprehensible results.

**Stronger AST Predicates** There are many cases where one would like to write transition predicates that cannot practically or maintainably be written using ASTs. An example from the EROS kernel is that we would like to know that every capability assignment is made to an "unhazarded" capability. If a capability is unhazarded, no cache state needs to be invalidated by the assignment. Unfortunately, there are a very large number of ASTs in which a capability assignment occurs. What we would like to write is an AST pattern matcher like:

`{ = { var x } { any }} s.t.  typeof($x)==Capability`

That is: "match any assignment to a variable where the type of that variable is capability." This sort of "meta-pattern" enhances the expressive power of the tool, but not its fundamental complexity.

**AST Instance Annotation** AST specification is extremely flexible, but it is limited to pattern matching unless the program can be annotated directly. On occasion, one wishes to write a transition rule that fires when a particular statement has completed. This requires AST instance matching rather than AST pattern matching, and is most effectively accomplished by labeling the statement in the source code. While C statement labels can be used for this purpose, many compilers complain about unused labels or labels that are not followed by a statement. Programmers respond to warnings by removing the offending labels. Our sense is that MOPS-specific syntactic annotation comments would be more robust for this purpose than statement labels.

**Typestate** Typestate [15, 16] is a property that tracks the evolution of dynamic type as a function of control flow. For example, the cache status of an EROS process is either encached, decached, or unknown. Certain operations on processes are safe only if the process is cached. This is not a property that can readily be captured by the static type (`Process`), but can be tracked as a mapping of the form:
(variable, ProgramCounter) $\rightarrow$ typestatesof(typeof(variable))
A great many of the properties we want to check can be expressed as typestate properties. In contrast to global control-flow properties, typestate can be sensibly expressed in the source code using syntactic comments. We can annotate each type declaration with its known type states, and annotate procedures with the typestate transitions or preconditions that they establish.

**Liveness Analysis and Value Flow** MOPS pattern variables provide a coarse approximation to reaching definitions. In practice, we would like to know not only that the same variable was passed, but that its value has not changed. More generally, we would like to have value flow analysis to detect distinct value continuity across FSA transitions.

Collectively, these enhancements would not bring MOPS to the full power of a theorem proving system, and there are many properties that the enhanced MOPS would remain unable to check. They would, however, substantially improve the utility of MOPS for assurance purposes.

## 7. RELATED WORK

Formal verification is an objective approach to demonstrate the correlation between a software design and its implementation. In this approach, the user specifies the behavior of a program, and the verification tool checks if the program satisfies the specification. LARCH [19] is such a tool. Another example is the B-Method [20], which provides a notation, a method, and a toolkit for requirement modeling, software interface specification, software design, implementation and maintenance. Although these tools can lead to astonishingly strong code, the work to create the specification is often heavy, and the skills needed are beyond most software programmers.

Light-weight formal verification has been proposed to overcome the complexity and scalability problems in full formal verification. Instead of creating a precise but complex specification of the program, in light-weight verification the user selects a set of simple properties that are easy to specify and computationally cheap to verify. As such, light-weight verification tools can afford to use relatively simple and scalable machinery. Splint [11] is a tool for statically checking C programs for a pre-defined set of common coding errors. Since it checks only those properties that were envisioned by the tool developer, it cannot be used to check application-specific properties.

SLAM [5] uses software model checking to verify user-specifiable temporal safety properties in programs. It iteratively finds error traces using a model checker, rejects infeasible traces using a theorem prover, and refines a boolean abstraction of the source program. BLAST [1], a software model checker similar to SLAM, uses lazy abstraction to reduce unnecessary abstraction refinement. ESP [21] is a tool for verifying temporal safety properties in C/C++ programs. It uses a global context sensitive, control flow insensitive analysis in the first phase and an inter-procedural, context sensitive dataflow analysis in the second phase. Among these tools, SLAM/BLAST are the most precise and least scalable, MOPS is the most scalable and least precise, and ESP is between SLAM/BLAST and MOPS in precision and scalability. SLAM, BLAST, and ESP might satisfy our need for typestate (Section 6), but SLAM and ESP are publicly unavailable and BLAST is not mature enough. CMC [22] model checks a system for erratic behaviors. The state of the system is the union of the states of all its processes along with the contents of the shared memory, and the state of each process consists of its global variables, heap, stack, and context registers. CMC handles its state explosion problem by using hash tables and heuristics, which effectively compromises soundness.

Rather than focus on temporal safety properties, some tools are concerned mainly with data flow properties. Koved et al. used context sensitive, flow sensitive, inter-procedural data flow analysis to compute access rights requirement in Java with optimizations to keep the analysis tractable [23]. CQUAL [24] is a type-based analysis tool that provides a mechanism for specifying and checking properties of C programs. It has been used to detect format string vulnerabilities [25] and to verify authorization hook placement in the Linux Security Model framework [4], which are examples of the development of sound analysis for verification of particular security properties. The application of CQUAL, however, is limited by its flow insensitivity. CQUAL may also satisfy our need for typestate. Extended Static Checker for Java (ESC/Java) [26] uses verification-condition generation and automatic theorem-proving techniques to find common programming errors. Compared to MOPS, ESC/Java requires the user to annotate programs to express their design decisions, which makes it harder to use and maintain in the software development process.

Aside from light-weight verification tools, there are light-weight bug finding tools. They do not attempt to verify the absence of bugs in a program; rather they find bugs in a program at best-effort. MC [12] is such a tool, which checks for rule violations in operating systems using meta-level compilation to write system-specific compiler extensions. Although MC has been used successfully to find many bugs, it does not provide assurance for any property. Furthermore, MC is not publicly available.

Specification and verification trails are usually considered proprietary data. Roger Schell, for example, views the trail of evidence supporting the assurance results for the Blacker kernel [27] as a proprietary data, not only because of what it reveals about the Blacker kernel, but because of what it reveals about how to achieve such verifications [28].

Perhaps the best documented effort to verify properties about a substantial software system is the verification work on PSOS [29, 30]. The PSOS verification method relies on a hierarchical structuring of the system design and its implementation followed by a theorem-proving verification of properties across the layers of the design. While this method generates justifiably higher confidence than the one used by MOPS, the effort required is substantially larger and the confidence of full verification is not required in all applications. At some threshold, bounding the maintenance cost of assurance over the course of the software life cycle becomes more important than achieving a higher degree of assurance. In such situations, lighter methods such as MOPS may provide a more effective cost/benefit trade-off.

# 8. CONCLUSION

In this paper we have reported on our use of the MOPS static checker to verify selected temporal safety properties on the EROS operating system kernel. In the process, we have both demonstrated the utility of the technique and arrived at greater confidence in the design and implementation of the EROS system.

In order to be effective as a tool for preventing the introduction of temporal safety errors, a verification technique must meet several requirements:

- It must be sufficiently sound (and must document clearly the conditions under which it may become unsound). The tool must not fail silently.
- Its specifications must be expressed in a form that typical programmers can write.
- It must not require invasive changes to the code base.
- It must be efficient enough to be incorporated into the normal build process.

The MOPS checker satisfies all of these properties.

Pragmatically, the cost, complexity, and time of verification must be balanced against developer utility and confidence gained. MOPS offers a reasonable trade-off in this cost/benefit continuum. It builds on specification techniques that are already known to programmers (FSAs) and reports errors in a form that programmers understand (traces). At the same time, the fact that safety properties can be formalized as finite state automata provides confidence that the system has been designed and implemented in a structured, principled, and robust way. Initial utility can be obtained by developers in hours, and substantial end-to-end checks in a few weeks. Once constructed, specifications provide a continuing guard against the introduction of temporal safety errors. No complex training in theorem proving is required.

Based on our experiences in this experiment, we suggest that checking of this kind should be incorporated more broadly into the development processes of critical software systems.

# 9. ACKNOWLEDGMENTS

The original EROS architecture was closely derived from that of KeyKOS, whose principal architects and implementors include Norman Hardy, Charlie Landau, and William Frantz. Each of these individuals has participated in and encouraged work on the EROS system.

David Wagner initiated the MOPS project, suggested using MOPS on EROS, and provided lots of insightful discussions and feedback. This work would not be possible without his help. Robert Johnson helped implement generic pattern variables in MOPS.

# 10. REFERENCES


[1] Thomas A. Henziger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Gregoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code. In *Proc. 14th International Conference on Computer-Aided Verification*, pages 526–538, 2002.

[2] Junfeng Yang, Ted Kremenek, Yichen Xie, and Dawson Engler. MECA: an extensible, expressive system and language for statically checking security properties. In *Proc. 10th ACM Conference on Computer and Communications Security*, 2003.

[3] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proc. 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, October 2003.

[4] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using CQUAL for static analysis of authorization hook placement. In *Proceedings of the Eleventh USENIX Security Symposium*, August 2002.

[5] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL '02: Proceedings of the ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, 2002.

[6] Hao Chen and David Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, 2002.

[7] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of C code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, San Diego, CA, 2004.

[8] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 170–185, Kiawah Island Resort, near Charleston, SC, USA, December 1999. ACM.

[9] J. S. Shapiro and S. Weber. Verifying the EROS confinement mechanism. In *Proc. 2000 IEEE Symposium on Security and Privacy*, pages 166–176, Oakland, CA, USA, 2000.

[10] Norman Hardy. The KeyKOS architecture. *Operating Systems Review*, 19(4):8–25, October 1985.

[11] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1), January 2002.

[12] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[13] J. S. Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, Philadelphia, PA 19104, 1999.

[14] Bryan Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullmann. Interface and execution models in the fluke kernel. In *Proc. 3rd Symposium on Operating System Design and Implementation*, pages 101–115, February 1999.

[15] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. on Software Engineering*, (1):157–171, January 1986.

[16] Robert E. Strom and Daniel M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE Trans. on Software Engineering*, (5):478–485, May 1993.

[17] David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. In *Proc. USENIX Symposium on Operating Systems Design and Implementation*, pages 179–193, 1994.

[18] Key Logic, Inc. *GNOSIS Design Documentation*, 1990.

[19] John V. Guttag, James J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *LARCH: Languages and Tools for Formal Specification*. Springer-Verlag, New York, NY, 1993.

[20] Emil Sekerinski and Kaisa Sere, editors. *Program Development by Refinement*. Springer, 1999.

[21] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[22] Madanlal Musuvathi, David Y.W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI '02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.

[23] Larry Koved, Marco Pistoia, and Aaron Kershenbaum. Access rights analysis for Java. In *Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2002.

[24] Jeffrey Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, May 1999.

[25] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[26] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, 2002.

[27] Roger R. Schell. Evaluating security properties of computer systems. In *Proc. 1983 IEEE Symposium on Security and Privacy*, pages 89–95, 1983.

[28] Roger R. Schell. *Evidence Trails as Proprietary Data*, 2002. Personal communication.

[29] R. Feiertag and P. Neumann. The foundations of a provably secure operating system (PSOS). In *Proc. 1979 National Computer Conference*, 1979.

[30] P.G. Neumann and R.J. Feiertag. PSOS revisited. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003), Classic Papers section*, Las Vegas, Nevada, December 2003.