Submitted to the *2004 IEEE Symposium on Security and Privacy*,
Oakland, CA 2004

Exploring Static Checking for Software Assurance

SRL Technical Report SRL-2003-06

Hao Chen
hchen@cs.berkeley.edu
Computer Science Division
U.C. Berkeley

Jonathan S. Shapiro
shap@cs.jhu.edu
Systems Research Laboratory
Department of Computer Science
Johns Hopkins University

November 4, 2003

Abstract

A key missing link in the creation of practically secure 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 a small number of bugs in the EROS implementation.

Current commodity assurance processes have no objectively verifiable benefit, while the formal verification needed for high assurance evaluation is expensive and requires specialized expertise. To date, no effective middle ground solution has emerged. 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.

1 Introduction

Robust software systems are difficult to design and maintain, and exceptionally difficult to assure. While seven levels of assurance (EAL1 through EAL7) are defined by current assurance standards [18], these levels can be divided qualitatively into only two groups: informal and formal. The informal assurance classifications (EAL1 through EAL5) differ in their level of detail, but all of the empirical evidence to date suggests that *all* of these assurance levels are have limited practical value [27]. Even if deployed correctly, no informally assured system today is likely to resist a concerted attempt at penetration for more than 24 hours in the field under normal commercial deployment scenarios.

Current higher assurance approaches do not offer a viable alternative. The gap between EAL6 (semi-formal) and EAL7 (formal) is in practice illusory: once the developer undertakes the effort to describe a system rigorously, formal notation is actually easier to use than semi-formal.[1] Since model checking techniques do not scale to the number of states required for a fully elaborated system specification, the requirement for "formal specification" has been taken to mean that the invariants of the system must be demonstrated using theorem proving techniques [15, 4]. While theorem proving oriented formal specification and verification techniques lead to astonishingly robust code, both are beyond the skills of all but a few experts. There are probably more than 10 such experts in the world today; there are certainly fewer than 100. The need for middle ground techniques that software developers can comprehend and maintain is acute.

Lightweight program analysis tools offer an alternative. However, many of them are not suitable for our task in hand. Dynamic (run-time) tools are insufficient because they only check the program paths that are executed during testing, which will likely miss a lot of other traces in the program. In comparison, static (compile-time) tools offer much better coverage of program paths. However, some static tools are designed

[1] This is our own opinion, but it also appears to be the consensus view of experienced high-assurance evaluators.

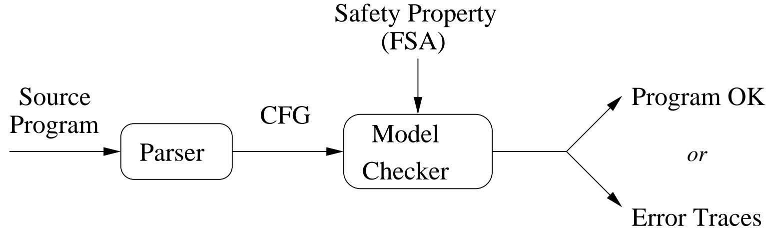


Figure 1: System Diagram of MOPS

to identify particular properties that were envisioned by the tool developer [12, 11, 8, 10]. Some general tools supporting user-specified properties, such as MC [9], 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 reduce the comprehensibility or maintainability of the code. Experience with annotation-driven checking tools [12] 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.

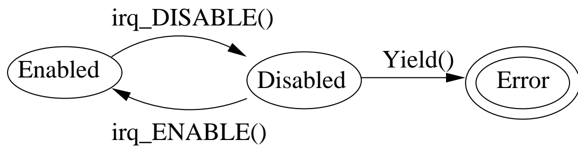
In this paper we report on an exploratory effort to apply application-specific software model checking as a middle ground approach for software assurance. We view the system as a collection of interacting state machines that collectively ensure end to end behavior. The transitions of these state machines can be expressed as temporal safety properties that can be exhaustively checked using a sound model checker. Rather than check all properties of the target system simultaneously, we check the properties individually using the MOPS static checking tool [5]. A key property of this approach is that the specifications (finite state machines) can be written by ordinary programmers and the results (traces) are readily comprehensible. 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.

2 Overview of MOPS

MOPS is a static (compile-time) analysis tool that checks if a program may violate selected safety properties [5]. The properties that MOPS checks are temporal safety properties, which require that programs perform certain operations in defined sequence. Many application security properties can be expressed as this sort of temporal safety property. The current implementation checks control flow properties augmented by local dataflow analysis.

MOPS consists of a parser and a model checker (Figure 1). The parser compiles a C source program into a Control Flow Graph (CFG). The model checker takes as input a CFG and a safety property described in a Finite State Automaton (FSA) by the MOPS user. Then, the model checker decides if the program may violate the property. If so, MOPS provides error traces, program paths that violate the property, to help the user identify program errors.

Conceptually, MOPS "compiles" the original source program into pseudo-instructions. Each of these instruction completes the execution of some abstract syntax tree (AST) node in the source program. When these instructions are simulated, they "emit" the identity of the corresponding AST nodes (the simulation does not consider value analysis). Thus, each possible trace of the program produces a string over the alphabet of AST nodes. These strings are then checked by using them as input to a finite-state automaton whose states and transition rules express the temporal constraint. In practice, the trace emission and the FSA check are fused, allowing the checker to eliminate non-terminating loops and unbounded recursions. It is



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

sufficient to simulate each "basic block" of the program once for each FSA state in which that basic block is executed by the program. Simulation along a given trace halts when that trace enters a final (error) state of the FSA. The checker is a pushdown model checker: it matches each function return with its call site.

For example, MOPS might be used to check the following property: the program should not call `Yield()` when 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 (see Section 4.1 for details on this property). The kernel may enable an interrupt by calling `irq_ENABLE` and disable an interrupt by calling `irq_DISABLE`. The MOPS user describes a property by a Finite State Automaton (FSA). Figure 2(a) shows an FSA describing the above property, and Figure 2(b) shows a code fragment that violates this property.

In large software systems, invariants of this sort 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 attempt 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]

2.1 Soundness

MOPS strives for soundness and scalability. A sound tool will not overlook any violations of the safety property in the program. In the current state of the art, however, perfect soundness results in poor scalability,

[2] Or, of course, because the program design lacked any model of correctness in the first place. Strong assertion appears to be the preferred means of establishing assurance in such systems. We are skeptical that formal methods can help.

i.e., the inability to check large programs. Since MOPS is designed to be a practical tool for checking safety properties on large programs, it tries to strike a balance between soundness and scalability. To strive for soundness, MOPS is path sensitive, i.e., it follows every path in the program (including arbitrary number of iterations of loops) except a few minor cases discussed below. MOPS is also context sensitive, i.e., MOPS 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. It, however, implements limited local data flow analysis so that it recognizes the same variable `x` in different expressions such as `x=open()` and `close(x)`. 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 assumptions:

- The program is single-threaded.
- The program is memory safe, e.g., no buffer overruns.
- The program is written in standard compliant C, augmented by selected GNU C extensions. MOPS does not understand inline assembly code, but has features that allow us to treat assembly functions as "external" procedures.
- The program does not violate the soundness assumptions required by the 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 `open()` and `close()`.

The current version 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()`). Although these may cause unsoundness, they are implementation limitations rather than intrinsic difficulties. From an analysis perspective, they do suggest that language-supported exception handling strongly improves our ability to analyze programs that do non-local returns. As it happens, the EROS kernel does not make use of `longjmp()`.[3]

Finally, MOPS chooses some particular resolution of control flow order where the C specification is ambiguous. For example, the C standard [19] does not specify the order of argument evaluation for functions or operators. MOPS currently resolves this by choosing the evaluation order actually used by our compiler. Programs that rely on the ordering of control flow in such computations are not standard compliant C programs. This limitation is a prototype limitation rather than an inherent limitation in the modeling technique.

2.2 Error Reporting

Since no static analysis tool for verifying temporal safety properties can be both sound and complete, MOPS is incomplete in that MOPS may generate *false positive traces*, i.e., program traces that are either infeasible or that do not violate the property. In its first implementation, MOPS could only generate one error trace

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

for each violated property. Because of this restriction, the presence of a false positive effectively prevented reporting of further errors. During the course of the experiment reported here, we augmented MOPS to report *all* error traces in a single run.

Unfortunately, large numbers of traces can easily overwhelm the user. A single programming error can cause many, sometimes infinitely many, traces, so it is undesirable, and sometimes impractical, to show all of them to the user. We would like MOPS to show only one error trace for each source of error. To implement this, the model checker clusters the error traces so that all the error traces in one cluster reflect the same source of error. Then, for each cluster MOPS selects a shortest error trace from that cluster and shows it to the user. This has two benefits: (1) it does not miss any source of errors because each error is represented by a cluster; and (2) it greatly reduces the number of error traces that the user has to review. This improvement provided orders of magnitude reduction in the number of error traces the user was forced to examine.

A subtle failure to report all errors remains in the tool today: because simulation halts when an error state is entered, it is possible for one error to mask a subsequent error in the same trace. In practice, this has little impact: first, as real errors are fixed, further errors are unmasked in subsequent executions of the tool; second, the lack of data-flow-aware path-sensitive analysis tends to make masked errors rare. False accusals result in practice from the fact that MOPS does not perform conditional liveness analysis [30]. Within the idioms of real programs, this analytic weakness tends to ensure that there exists *some* path that continues execution past the false positive. For example, in the sequence:

```
if (x) lock();
statement;
if (x) unlock();
```

MOPS assumes that there exists a valid trace that bypasses *both* conditionals. As a result, subsequent errors are not masked in this example. A more precise model checking tool might need to deal with this problem, but the problem has not affected us yet.

2.3 Multiple Entry Points

While C application programs start from the `main` function, operating systems tend to have multiple entry points that are reachable from assembly language exception handlers. To support analysis in this setting, we enhanced MOPS to accept specification of multiple entry points. Later, we noticed that this enabled us to introduce a workaround that partially resolves the problem of function pointers: while MOPS does not attempt to simulate indirect function calls, it is possible to add the destination functions as entry points.

We note that indirect function simulation is not an insurmountable difficulty in practice. It would not be difficult using value flow analysis to detect field assignments to function pointer fields and construct a mapping from these fields to the possible destination functions (points-to analysis). In languages supporting inheritance, virtual functions can be handled even more straightforwardly (receiver class analysis). MOPS does not implement either feature today.

2.4 External Functions

Applications frequently make use of library functions or operating system services whose source code is unavailable to the checker. To support analysis of such applications, MOPS enables us to capture the distinction between application programs and their environment by providing two ways of matching the start and end of functions: for functions whose source code is available, the `function_entry` and `function_exit` AST types match the first and (respectively) last instructions of the function. For functions whose source code is *not* available, the `function_call` AST type matches the call site of the specified function. The distinction between function call and function entry exists to match the distinction between internal and external program interactions. For some properties, it is important to trace the internal flow of the program through the source code of a function. For others, the function call crosses a logical boundary between

modules or subsystems, and should not be traced when validating the property at hand. Within a single state machine, MOPS safeguards the user from using both `function_call` and `function_entry` for the same function because these two semantics are mutually exclusive.

While operating systems do not tend to link binary-only code, they *do* have assembly language procedures that MOPS is unable to analyze. In an interrupt-style kernel, there are also procedures that do not exit, because they abandon the current path of execution within the kernel. The distinction between function call and function entry allowed us to express transitions associated with these assembly language routines, and also allowed us to express non-returning procedures by transitioning to dead-end states in the FSA.

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 [28, 26].

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 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 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 permissably 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 [6]. 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 [26]. The software-management strategy demands data structures for dependency tracking. The code that maintains these data structures naturally follows control flow invariants that can be statically checked by model checking.

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 [31, 30]. 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, it is also somewhat frustrating: invariant discipline is very effective in resisting bugs, and there were relatively few to find.

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. All kernel system calls are expected to be invoked with interrupts enabled.

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 implemented five distinct disabled states, and used these to simulate up to five levels of nesting. We additionally provided a sixth disable state as a guard state to validate that five were enough – the guard state is an error state. The need for five 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 [30].

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 something 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()`. There is a known exceptional case in the timer logic.

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 [29]. 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 the directness of correspondence between this property and the previous formal verification work on the EROS confinement mechanism [29].

Sadly or happily (depending on which author you ask), we found a bug in the course of this analysis. The EROS kernel is midway through a transition to more dynamic heap management, and there exists one system service call used by user-mode drivers that allocates heap memory from a finite kernel pool. This allocation can fail, and the implementation called `Yield()` if it does. Actually, there were two distinct bugs revealed by this test:

- The call to `malloc()` occurred after the commit point.
- The `malloc()` function called `Yield()`, which is an error because memory returned by previous calls to malloc may need to be released. EROS has a generalized strategy for releasing other resources when a `Yield()` occurs, but pointers to successively malloc'd regions need to be handled with care. The invariant failure revealed that we had entirely failed to consider this challenge when we made the decision 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. 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 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. The second implies that every call to `Sleep()` must be followed at some later point by a call to `Yield()`.

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 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()`.

In our original specification of this property, we captured it as described above. In the process 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.[4] 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` that is cleared on entry to the kernel and set whenever a page table entry is invalidated. This approach is conservative, but effective.

In defensive cases such as these, 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. 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 first encache the process

[4] 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.

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 "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.

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.

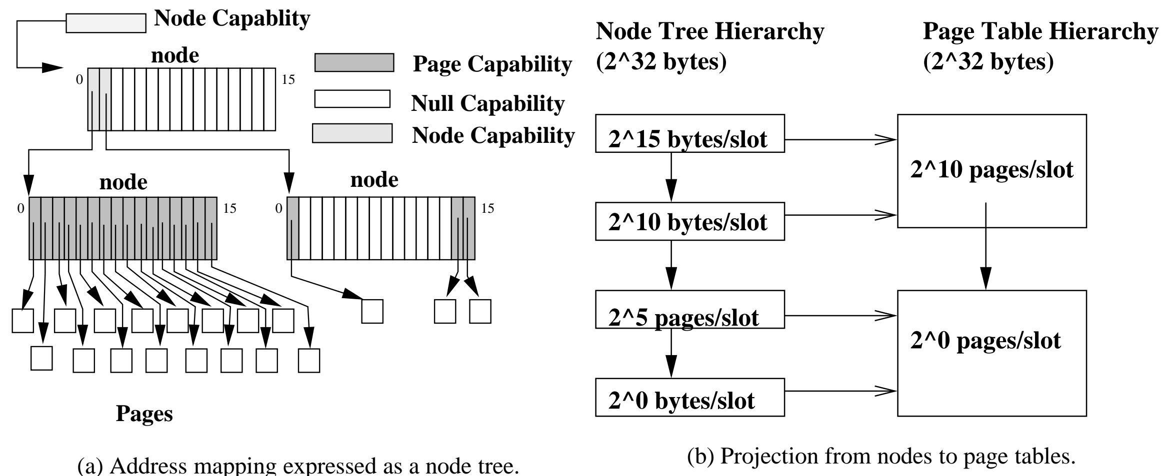
As expected, no errors were found by this check.

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.

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 (Figure 3(b)) corresponds to some path in the node tree (Figure 3(a)).

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 (Figure 3) of the first (the node tree) [26]. Because of the difference in tree arity, the correctness of the correspondence – never mind



(a) Address mapping expressed as a node tree.

(b) Projection from nodes to page tables.

Figure 3: Correspondence between node and page table trees.

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 everything can be reduced easily to temporal safety properties. The less obvious one is that *most* things can: of the many properties 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 this technique can do.

4.6 Performance and Usability

In the course of this experiment, our use of MOPS was iterative. Writing the initial property specification would take just a few minutes, but this property would be checked and found to produce errors because the specification was in some way incomplete. New states and transitions would be added to the FSA to address the deficiency, and the property would be re-run. Two or three iterations of this was generally sufficient to accurately capture the property. The properties described here were written during the course of roughly sixteen hours spread over several conference trips; a long attention span was not required. The paper describing them took longer to write than the properties themselves.

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 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.

Executing the checks is reasonably fast. The EROS kernel is approximately 60,000 lines of code. Producing the complete CFG for this code base approximately doubles the compilation time, because the CFG-generating compiler must be run for each source file that is compiled. Given the complete CFG, it takes

MOPS less than a minute to check any one of the properties described here. This is not fast enough to perform with every link step, but it is certainly fast enough to do before committing kernel changes into the source code repository.

5 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 [31, 30] 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:

$$(\text{variable}, \text{ProgramCounter}) \longrightarrow \text{typestatesof}(\text{typeof}(\text{variable}))$$

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.

6 Assurance

In the context of secure software, the purpose of verification is to establish confidence in both the software artifact and its design. An effective body of confidence metrics achieves two independent goals: specific and generic. Specific confidence is confidence that the test or verification has successfully checked the specified property. Soundness is an essential precondition for this. The absence of reported errors must not mean that the tool or technique failed silently.

From the standpoint of assurance, however, generic confidence may be more important. Generic confidence is confidence that the designers of the system adopted an invariant-oriented design approach, and that the developers are working faithfully to observe those invariants. We are aware of no real system with a mean time between failures (MTBF) of less than a year that incorporates invariant-based design. Conversely, there do not seem to be many systems with an MTBF *greater* than one year that are *not* designed and implemented using an invariant-driven process. As one proceeds through the process of specifying and checking properties, one very quickly discovers whether these underlying invariants exist in the system. Unprincipled designs are remarkably hard to specify.

Historically, the EROS system has a long track record of specifying invariants before implementing them. While these specifications have not been collected in a single place (and 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 [20]. In the course of this exploration, we found many points where our memory of this specification had drifted slightly, or our recollection of the details of the implementation had deviated in minor ways from the code. What was striking was not that these deviations existed, but that over 13 years of work on the current code base they had drifted so little. While we found bugs, *all* of these bugs were discovered in recent, experimental code. It remains to be seen how easily EROS can be penetrated once it is widely deployed, and we are certain that there will be several initial embarrassments as minor flaws in the kernel and critical utilities are discovered. We are, at this point, confident that the overall invariant structure of the system and the implementation of these invariants will hold up well.

The objective of assurance is confidence, which is an inherently subjective metric. A full verification at a given point in time can show that a system is credible at that instant with respect to the properties that have been checked. It also reveals something about the underlying system structure, and it is this structure that dictates how sustainable the system's security and robustness will be. We have found in this work that property-specific model checking similarly reveals the underlying system structure, and that it offers three key advantages over theorem proving methods:

- The specifications can be written by ordinary programmers.
- The resulting failures take the form of traces, which are readily understandable.
- The checking process is fast enough that it can be integrated into the development cycle, providing rapid feedback. An ordinary FSA-based specification is relatively robust against changes in the code.

Based on our experience in this exploration, we would strongly advocate domain-specific checking of temporal safety properties as a basis for confidence in establishing assurance.

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 [15, 4] is such a tool. 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. At an extreme, if the user tries to specify the complete behavior of the program, then the specification becomes as complex and error-prone as the code itself. Because formal

verification tools aim at verifying complex properties, they have to rely on heavy machinery, which severely limits their scalability.

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 [12, 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 [2, 3] uses software model checking to verify user-specifiable temporal safety properties in programs in an iterative process. During each iteration, a model checker determines the reachability of certain states in a boolean abstraction of the source program and a theorem prover verifies the path given by the model checker. If the path is infeasible, additional predicates are added and the process enters a new iteration. Compared to MOPS, SLAM is heavier and less scalable but more precise. BLAST [17], a software model checker similar to SLAM, uses lazy abstraction to reduce unnecessary abstraction refinement. BLAST is comparable to SLAM in scalability and precision. ESP [7] 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. ESP is between SLAM/BLAST and MOPS in scalability and precision. SLAM, BLAST, and ESP might satisfy our need for typestate (Section 5), but SLAM and ESP are publicly unavailable and BLAST is not mature enough. CMC [21] model checks a system for erratic behaviors. It models the system as a collection of interacting concurrent processes. The state of each process consists of its global variables, heap, stack, and context registers, and the state of the system is the union of the states of all its processes along with the contents of the shared memory. This approach easily leads to the state explosion problem, which CMC handles by using hash tables and heuristics, which effectively compromises soundness.

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 – they do not guarantee to find all the bugs in the program. MC [16] 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 [1], it is unsound and it does not state if it can be sound under any condition. Furthermore, MC is also 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 [24] 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 [25].

Perhaps the best documented effort to verify properties about a substantial software system is the verification work on PSOS [13, 23, 22]. 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 maintainence cost of assurance over the course of the software lifecycle 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 tradeoff.

8 Acknowledgments

While it has diverged in recent years, the original EROS architecture was closely derived from that of KeyKOS. No work derived from KeyKOS could be complete without acknowledging the principal architects

and implementors of that system: Norman Hardy, Charlie Landau, and William Frantz. Each of these individuals has participated in and encouraged work on the EROS system.

David Wagner at U.C. Berkeley initiated the MOPS project, identified several original algorithms in MOPS, and provided lots of insightful feedback. The work described here would not be possible without his assistance and help. Robert Johnson contributed the initial implementation of generic pattern variable in MOPS.

9 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 in the context of assurance, 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.
- Specifications must be expressed in a form that programmers can write.
- It must report results in a way that is comprehensible.
- It must not require invasive changes to the code base.

The MOPS checker satisfies all of these properties.

In the context of software assurance, the objective of formal methods is to establish confidence in both the design and the implementation of a system. Pragmatically, the cost, complexity, and time of verification must be balanced against the confidence gained. MOPS offers a "middle ground" 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 in hours, and substantial end-to-end confidence in a few weeks. No complex training in theorem proving is required. Based on our experiences in this experiment, we feel that checking of this kind should be accepted as a credible basis for confidence in the context of high assurance evaluations.

Source code for the EROS system, including the property specifications described here, can be obtained from `www.eros-os.org`. Source code for the version of MOPS used here is also included in the EROS source tree.

References

[1] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of IEEE Security and Privacy 2002*, 2002.

[2] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001, Workshop on Model Checking of Software*, 2001.

[3] T. Ball and S. 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.

[4] J. Barnes. *High Integrity Software: The Spark Approach to Safety and Security*. Addison Wesley, 2003.

[5] H. Chen and D. 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.

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

[7] M. Das, S. Lerner, and M. 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.

[8] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proc. 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, Oct. 2003.

[9] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific programmer-written compiler extentions. In *Proc. 4th Symposium on Operating Systems Design and Implementation*, pages 57–72, San Diego, CA, Oct. 2000.

[10] D. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Proc. 18th ACM Symposium on Operating Systems Principles*, pages 57–72, Lake Louise, Banff, Alberta, CA, Oct. 2001.

[11] D. Evans. Static detection of dynamic memory errors. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 44–53, Philadelphia, PA, May 1996.

[12] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1), Jan. 2002.

[13] R. Feiertag and P. Neumann. The foundations of a provably secure operating system (psos). In *Proc. 1979 National Computer Conference*, pages 329–334, 1979.

[14] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann. Interface and execution models in the fluke kernel. In *Proc. 3rd Symposium on Operating System Design and Implementation*, pages 101–115, Feb. 1999.

[15] J. V. Guttag, J. 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.

[16] S. Hallem, B. Chelf, Y. Xie, and D. 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*, 2002.

[17] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL '02: Proceedings of the ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, 2002.

[18] *Common Criteria for Information Technology Security*. International Standards Organization, 1998. International Standard ISO/IS 15408, Final Committee Draft, version 2.0.

[19] *International Standard ISO/IEC 9899:1999 (Programming Languages - C)*. International Standards Organization, 1999.

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

[21] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. 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.

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

[23] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. Technical Report Report CSL-116, Computer Science Laboratory, may 1980.

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

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

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

[27] J. S. Shapiro. Understanding the windows EAL4 evaluation. *IEEE Computer*, (2):102–105, Feb. 2003.

[28] J. S. Shapiro, J. M. Smith, and D. 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, Dec. 1999. ACM.

[29] 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.

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

[31] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. on Software Engineering*, (1):157–171, Jan. 1986.