DOI:10.1145/1743546.1743574

seL4: Formal Verification of an Operating-System Kernel

By Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood

Abstract

We report on the formal, machine-checked verification of the seL4 microkernel from an abstract specification down to its C implementation. We assume correctness of compiler, assembly code, hardware, and boot code.

seL4 is a third-generation microkernel of L4 provenance, comprising 8700 lines of C and 600 lines of assembler. Its performance is comparable to other high-performance L4 kernels.

We prove that the implementation always strictly follows our high-level abstract specification of kernel behavior. This encompasses traditional design and implementation safety properties such as that the kernel will never crash, and it will never perform an unsafe operation. It also implies much more: we can predict precisely how the kernel will behave in every possible situation.

1. INTRODUCTION

Almost every paper on formal verification starts with the observation that software complexity is increasing, that this leads to errors, and that this is a problem for mission and safety critical software. We agree, as do most.

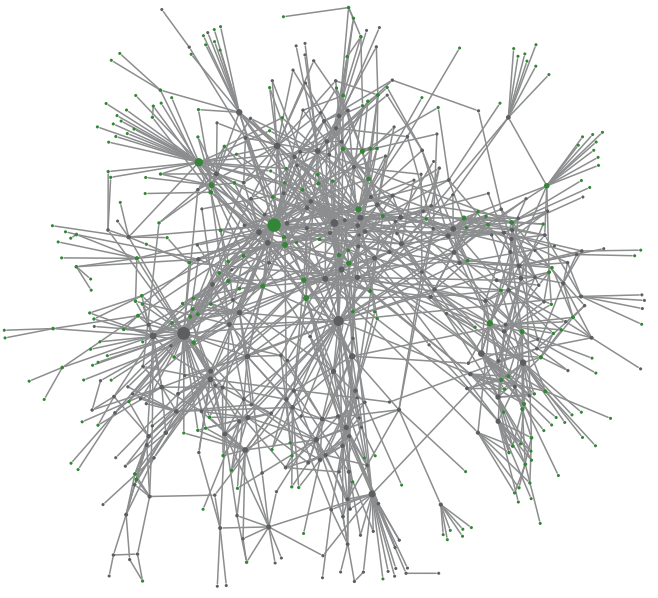
Here, we report on the full formal verification of a critical system from a high-level model down to very low-level C code. We do not pretend that this solves all of the software complexity or error problems. We do think that our approach will work for similar systems. The main message we wish to convey is that a formally verified commercial-grade, general-purpose microkernel now exists, and that formal verification is possible and feasible on code sizes of about 10,000 lines of C. It is not cheap; we spent significant effort on the verification, but it appears cost-effective and more affordable than other methods that achieve lower degrees of trustworthiness.

To build a truly trustworthy system, one needs to start at the operating system (OS) and the most critical part of the OS is its *kernel*. The kernel is defined as the software that executes in the privileged mode of the hardware, meaning that there can be no protection from faults occurring in the kernel, and every single bug can potentially cause arbitrary damage. The kernel is a mandatory part of a system's *trusted computing base* (TCB)—the part of the system that can bypass security.[10] Minimizing this TCB is the core concept behind *microkernels*, an idea that goes back 40 years.

A microkernel, as opposed to the more traditional *monolithic* design of contemporary mainstream OS kernels, is reduced to just the bare minimum of code wrapping hardware mechanisms and needing to run in privileged mode. All OS services are then implemented as normal programs, running entirely in (unprivileged) user mode, and therefore can potentially be excluded from the TCB. Previous implementations of microkernels resulted in communication overheads that made them unattractive compared to monolithic kernels. Modern design and implementation techniques have managed to reduced this overhead to very competitive limits.

A microkernel makes the trustworthiness problem more tractable. A well-designed high-performance microkernel, such as the various representatives of the L4 microkernel family, consists of the order of 10,000 lines of code (10 kloc). This radical reduction to a bare minimum comes with a price in complexity. It results in a high degree of interdependency between different parts of the kernel, as indicated in Figure 1. Despite this increased complexity in low-level code, we have demonstrated that with modern techniques and careful design, an OS microkernel is entirely within the realm of full formal verification.

Figure 1. Call graph of the seL4 microkernel. Vertices represent functions, and edges invocations.



The original version of this paper was published in the *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles, Oct.* 2009.

Formal verification of software refers to the application of mathematical proof techniques to establish properties about programs. Formal verification can cover not just all lines of code or all decisions in a program, but all possible behaviors for all possible inputs. For example, the very simple fragment of C code `if (x<y)z=x/y else z=y/x` for `x, y,` and `z` being `int` tested with `x=4, y=2` and `x=8, y=16`, results in full code coverage: every line is executed at least once, and every branch of every condition is taken at least once. Yet, there are still two potential bugs remaining. Of course, any human tester will find inputs such as `x=0, y=-1` and `x=-1, y=0` that expose the bugs, but for bigger programs it is infeasible to be sure of completeness. This is what formal verification can achieve.

The approach we use is interactive, machine-assisted, and machine-checked proof. Specifically, we use the theorem prover Isabelle/HOL.[8] Interactive theorem proving requires human intervention and creativity to construct and guide the proof. It has the advantage that it is not constrained to specific properties or finite, feasible state spaces. We have proved the *functional correctness* of the seL4 microkernel, a secure embedded microkernel of the L4[6] family. This means, we have proved mathematically that the implementation of seL4 always strictly follows our high-level abstract specification of kernel behavior. This property is stronger and more precise than what automated techniques like model checking, static analysis, or kernel implementations in type-safe languages can achieve. We not only analyze specific aspects of the kernel, such as safe execution, but also provide a full specification and proof for the kernel's precise behavior.

In the following, we describe what the implications of the proof are, how the kernel was designed for verification, what the verification itself entailed and what its assumptions are, and finally what effort it cost us.

2. IMPLICATIONS

In a sense, functional correctness is one of the strongest properties to prove about a system. Once we have proved functional correctness with respect to a model, we can use this model to establish further properties instead of having to reason directly about the code. For instance, we prove that every system call terminates by looking at the model instead of the code. However, there are some security-relevant properties, such as transmission of information via covert channels, for which the model may not be precise enough.

So our proof does not mean that seL4 is secure for all purposes. We proved that seL4 is functionally correct. *Secure* would first need a formal definition and depends on the application. Taken seriously, security is a whole-system question, including the system's human components.

Even without proving specific security properties on top, a functional correctness proof already has interesting implications for security. If the assumptions listed in Section 4.5 are true, then in seL4 there will be:

No code injection attacks: If we always know precisely what the system does and if the spec does not explicitly allow it, then we can never have any foreign code executing as part of seL4.

No buffer overflows: This is mainly a classic vector for code injection, but buffer overflows may also inject unwanted data and influence kernel behavior that way. We prove that all array accesses are within bounds and we prove that all pointer accesses are well typed, even if they go via casts to void or address arithmetic.

No NULL pointer access: NULL pointer bugs can allow local privilege escalation and execution of arbitrary code in kernel mode.[9] Absence of NULL pointer dereference is a direct proof obligation for us for every pointer access.

No ill-typed pointer access: Even though the kernel code deliberately breaks C type safety for efficiency at some points, in order to predict that the system behaves according to specification, we prove that circumventing the type system is safe at all these points.

No memory leaks and no memory freed that is still in use. This is not purely a consequence of the proof itself. Much of the design of seL4 was focused on explicit memory management. Users may run out of memory, but the kernel never will.

No nontermination: We have proved that all kernel calls terminate. This means the kernel will never suddenly freeze and not return from a system call. This does not mean that the whole system will never freeze. It is still possible to write bad device drivers and bad applications, but set up correctly, a supervisor process can always stay in control of the rest of the system.

No arithmetic or other exceptions: The C standard defines a long list of things that can go wrong and that should be avoided: shifting machine words by a too-large amount, dividing by zero, etc. We proved explicitly that none of these occur, including the absence of errors due to overflows in integer arithmetic.

No unchecked user arguments: All user input is checked and validated. If the kernel receives garbage or malicious arguments it will respond with the specified error messages, not with crashes. Of course, the kernel will allow a thread to kill itself if that thread has sufficient capabilities. It will never allow anything to crash the kernel, though.

Many of these are general security traits that are good to have for any kind of system. We have also proved a large number of properties that are specific to seL4. We have proved them about the kernel design and specification. With functional correctness, we know they are true about the code as well. Some examples are:

Aligned objects: Two simple low-level invariants of the kernel are: all objects are aligned to their size, and no two objects overlap in memory. This makes comparing memory regions for objects very simple and efficient.

Well-formed data structures: Lists, doubly linked, singly linked, with and without additional information, are a pet topic of formal verification. These data structures also occur in seL4 and we proved the usual properties: lists are not circular when they should not be, back pointers point to the right nodes, insertion, deletion etc., work as expected.

Algorithmic invariants: Many optimizations rely on certain properties being always true, so specific checks can be left out or can be replaced by other, more efficient checks. A simple example is that the distinguished idle thread is always in thread state *idle* and therefore can never be blocked or

otherwise waiting for I/O. This can be used to remove checks in the code paths that deal with the idle thread.

Correct book-keeping: The seL4 kernel has an explicit user-visible concept of keeping track of memory, who has access to it, who access was delegated to, and what needs to be done if a privileged process wants to revoke access from delegates. It is the central mechanism for reusing memory in seL4. The data structure that backs this concept is correspondingly complex and its implications reach into almost all aspects of the kernel. For instance, we proved that if a live object exists anywhere in memory, then there exists an explicit capability node in this data structure that covers the object. And if such a capability exists, then it exists in the proper place in the data structure and has the right relationship towards parents, siblings, and descendants within. If an object is live (may be mentioned in other objects anywhere in the system) then the object itself together with that capability must have recorded enough information to reach all objects that refer to it (directly or indirectly). Together with a whole host of further invariants, these properties allow the kernel code to reduce the complex, system-global test whether a region of memory is mentioned anywhere else in the system to a quick, local pointer comparison.

We have proved about 80 such invariants on the executable specification such that they directly transfer to the data structures used in the C program.

A verification like this is not an absolute guarantee. The key condition in all this is *if the assumptions are true*. To attack any of these properties, this is where one would have to look. What the proof really does is take 7500 lines of C code out of the equation. It reduces possible attacks and the human analysis necessary to guard against them to the assumptions and specification. It also is the basis for any formal analysis of systems running on top of the kernel or for further high-level analysis of the kernel itself.

3. KERNEL DESIGN FOR VERIFICATION

The challenge in designing a verifiable and usable kernel lies in reducing complexity to make verification easier while maintaining high performance.

To achieve these two objectives, we designed and implemented a microkernel from scratch. This kernel, called seL4, is a third-generation microkernel, based on L4 and influenced by EROS.[11] It is designed for practical deployment in embedded systems with high trustworthiness requirements. One of its innovations is completely explicit memory-management subject to policies defined at user level, even for kernel memory. All authority in seL4 is mediated by *capabilities*,[2] tokens identifying objects and conveying access rights.

We first briefly present the approach we used for a kernel/proof codesign process. Then we highlight the main design decisions we made to simplify the verification work.

3.1. Kernel/proof codesign process

One key idea in this project was bridging the gap between verifiability and performance by using an iterative approach to kernel design, based around an intermediate target that is readily accessible to both OS developers and formal methods practitioners. We used the functional language Haskell to provide a programming language for OS developers, while at the same time providing an artifact that can readily be reasoned about in the theorem proving tool: the design team wrote increasingly complete prototypes of the kernel in Haskell, exporting the system call interface via a hardware simulator to user-level binary code. The formal methods team imported this prototype into the theorem prover and used it as an intermediate executable specification. The approach aims at quickly iterating through design, prototype implementation, and formal model until convergence.

Despite its ability to run real user code, the Haskell kernel remains a prototype, as it does not satisfy our high-performance requirement. Furthermore, Haskell requires a significant run-time environment (much bigger than our kernel), and thus violates our requirement of a small TCB. We therefore translated the Haskell implementation *manually* into high-performance C code. An automatic translation (without proof) would have been possible, but we would have lost most opportunities to micro-optimize the kernel in order to meet our performance targets. We do not need to trust the translations into C and from Haskell into Isabelle—we formally verify the C code as it is seen by the compiler, gaining an end-to-end theorem between formal specification and the C semantics.

3.2. Design decisions

Global Variables and Side Effects: Use of global variables and functions with side effects is common in operating systems—mirroring properties of contemporary computer hardware and OS abstractions. Our verification techniques can deal routinely with side effects, but implicit state updates and complex use of the same global variable for different purposes make verification more difficult. This is not surprising: the higher the conceptual complexity, the higher the verification effort.

The deeper reason is that global variables usually require stating and proving invariant properties. For example, scheduler queues are global data structures frequently implemented as doubly linked lists. The corresponding invariant might state that all back links in the list point to the appropriate nodes and that all elements point to thread control blocks and that all active threads are in one of the scheduler queues.

Invariants are expensive because they need to be proved not only locally for the functions that directly manipulate the scheduler queue, but for the whole kernel—we have to show that no other pointer manipulation in the kernel destroys the list or its properties. This proof can be easy or hard, depending on how modularly the global variable is used.

Dealing with global variables was simplified by deriving the kernel implementation from Haskell, where side effects are explicit and drawn to the design team's attention.

Kernel Memory Management: The seL4 kernel uses a model of memory allocation that exports control of the in-kernel allocation to appropriately authorized applications. While this model is mostly motivated by the need for precise guarantees of memory consumption, it also benefits verification. The model pushes the policy for allocation outside the kernel, which means we only need to prove that the mechanism

works, not that the user-level policy makes sense. The mechanism works if it keeps kernel code and data structures safe from user access, if the virtual memory (VM) subsystem is fully controlled by the kernel interface via capabilities, and if it provides the necessary functionality for user level to manage its own VM policies.

Obviously, moving policy into user land does not change the fact that memory allocation is part of the TCB. It does mean, however, that memory allocation can be verified separately, and can rely on verified kernel properties.

The memory-management model gives free memory to the user-level manager in the form of regions tagged as *untyped*. The memory manager can split untyped regions and retype them into one of several kernel object types (one of them, *frame*, is for user-accessible memory); such operations create new capabilities. Object destruction converts a region back to untyped (and invalidates derived capabilities).

Before reusing a block of memory, all references to this memory must be invalidated. This involves either finding all outstanding capabilities to the object, or returning the object to the memory pool only when the last capability is deleted. Our kernel uses both approaches. In the first approach, a so-called capability derivation tree is used to find and invalidate all capabilities referring to a memory region. In the second approach, the capability derivation tree is used to ensure, with a check that is local in scope, that there are no system-wide dangling references. This is possible because all other kernel objects have further invariants on their own internal references that relate back to the existence of capabilities in this derivation tree.

Similar book-keeping would be necessary for a traditional *malloc/free* model in the kernel. The difference is that the complicated *free* case in our model is concentrated in one place, whereas otherwise it would be repeated numerous times over the code.

Concurrency and Nondeterminism: Concurrency is the execution of computation in parallel (in the case of multiple hardware processors), or by nondeterministic interleaving via a concurrency abstraction like threads. Reasoning about concurrent programs is hard, much harder than reasoning about sequential programs. For the time being, we limited the verification to a single-processor version of seL4.

In a uniprocessor kernel, concurrency can result from three sources: *yielding* of the processor from one thread to another, (synchronous) *exceptions* and (asynchronous) *interrupts*. Yielding can be synchronous, by an explicit handover, such as when blocking on a lock, or asynchronous, by preemption (but in a uniprocessor kernel, the latter can only happen as the result of an interrupt).

We limit the effect of all three by a kernel design which explicitly minimizes concurrency.

Exceptions are completely avoided, by ensuring that they never occur. For instance, we avoid virtual-memory exceptions by allocating all kernel data structures in a region of VM which is always guaranteed to be mapped to physical memory. System-call arguments are either passed in registers or through preregistered physical memory frames.

The complexity of synchronous *yield* we avoid by using an event-based kernel execution model, with a single kernel stack, and a mostly atomic application programming interface. This is aided by the traditional L4 model of system calls which are primitive and mostly short-running.

We minimize the effect of interrupts (and hence preemptions) by disabling interrupts during kernel execution. Again, this is aided by the L4 model of short system calls.

However, not all kernel operations can be guaranteed to be short; object destruction especially can require almost arbitrary execution time, so not allowing any interrupt processing during a system call would rule out the use of the kernel for real-time applications, undermining the goal of real-world deployability.

We ensure bounded interrupt latencies by the standard approach of introducing a few, carefully placed, *interrupt points*. On detection of a pending interrupt, the kernel explicitly returns through the function call stack to the kernel/user boundary and responds to the interrupt. It then restarts the original operation, including reestablishing all the preconditions for execution. As a result, we completely avoid concurrent execution in the kernel.

I/O: Interrupts are used by *device drivers* to affect I/O. L4 kernels traditionally implement device drivers as user-level programs, and seL4 is no different. Device interrupts are converted into messages to the user-level driver.

This approach removes a large amount of complexity from the kernel implementation (and the proof). The only exception is an in-kernel timer driver which generates timer ticks for scheduling, which is straightforward to deal with.

4. VERIFICATION OF seL4

This section gives an overview of the formal verification of seL4 in the theorem prover Isabelle/HOL.[8] The property we are proving is functional correctness. Formally, we are showing *refinement*: A refinement proof establishes a correspondence between a high-level (abstract) and a low-level (concrete, or *refined*) representation of a system.

The correspondence established by the refinement proof ensures that all Hoare logic properties of the abstract model also hold for the refined model. This means that if a security property is proved in Hoare logic about the abstract model (not all security properties can be), our refinement guarantees that the same property holds for the kernel source code. In this paper, we concentrate on the general functional correctness property. We have also modelled and proved the security of seL4's access-control system in Isabelle/HOL on a high level.[3]

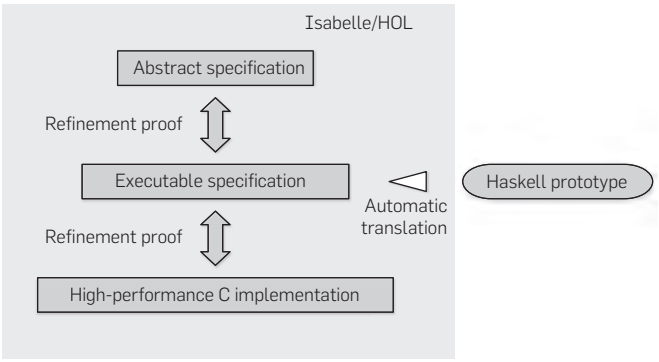
Figure 2 shows the specification layers used in the verification of seL4; they are related by formal proof. In the following sections we explain each layer in turn.

4.1. Abstract specification

The abstract level describes *what* the system does without saying *how* it is done. For all user-visible kernel operations, it describes the functional behavior that is expected from the system. All implementations that refine this specification will be binary compatible.

We precisely describe argument formats, encodings and error reporting, so, for instance, some of the C-level size restrictions become visible on this level. We model finite machine words, memory, and typed pointers explicitly.

Figure 2. The refinement layers in the verification of seL4.



Otherwise, the data structures used in this abstract specification are high level—essentially sets, lists, trees, functions, and records. We make use of nondeterminism in order to leave implementation choices to lower levels: if there are multiple correct results for an operation, this abstract layer would return all of them and make clear that there is a choice. The implementation is free to pick any one of them.

An example of this is scheduling. No scheduling policy is defined at the abstract level. Instead, the scheduler is modelled as a function picking *any* runnable thread that is active in the system *or* the idle thread. The Isabelle/HOL code for this is shown in Figure 3. The function `all_active_tcbs` returns the abstract set of all runnable threads in the system. Its implementation (not shown) is an abstract logical predicate over the whole system. The `select` statement picks any element of the set. The `OR` makes a nondeterministic choice between the first block and `switch_to_idle_thread`. The executable specification makes this choice more specific.

4.2. Executable specification

The purpose of the executable specification is to fill in the details left open at the abstract level and to specify how the kernel works (as opposed to what it does). While trying to avoid the messy specifics of how data structures and code are optimized in C, we reflect the fundamental restrictions in size and code structure that we expect from the hardware and the C implementation. For instance, we take care not to use more than 64 bits to represent capabilities, exploiting known alignment of pointers. We do not specify in which way this limited information is laid out in C.

The executable specification is deterministic; the only nondeterminism left is that of the underlying machine. All data structures are now explicit data types, records, and lists with straightforward, efficient implementations in C. For example the capability derivation tree of seL4, modelled as a tree on the abstract level, is now modelled as a doubly linked list with limited level information. It is manipulated explicitly with pointer-update operations.

Figure 4 shows part of the scheduler specification at this level. The additional complexity becomes apparent in the `chooseThread` function that is no longer merely a simple predicate, but rather an explicit search backed by data structures for priority queues. The specification fixes the behavior of the scheduler to a simple priority-based round-robin algorithm. It mentions that threads have time slices and it clarifies when the idle thread will be scheduled. Note that priority queues duplicate information that is already available (in the form of thread states), in order to make it available *efficiently*. They make it easy to find a runnable thread of high priority. The optimization will require us to prove that the duplicated information is consistent.

We have proved that the executable specification correctly implements the abstract specification. Because of its extreme level of detail, this proof alone already provides stronger design assurance than has been shown for any other general-purpose OS kernel.

4.3. C implementation

The most detailed layer in our verification is the C implementation. The translation from C into Isabelle is correctness-critical and we take great care to model the semantics of our C subset precisely and foundationally. *Precisely* means that we treat C semantics, types, and memory model as the C99 standard[4] prescribes, for instance, with architecture-dependent word size, padding of structs, type-unsafe casting of pointers, and arithmetic on addresses. As kernel programmers do, we make assumptions about the compiler (GCC) that go beyond the standard, and about the architecture

Figure 3. Isabelle/HOL code for scheduler at abstract level.

```
schedule ≡ do
  threads ← all_active_tcbs;
  thread ← select threads;
  switch_to_thread thread
od OR switch_to_idle_thread
```

Figure 4. Haskell code for schedule.

```
schedule = do
  action <- getSchedulerAction
  case action of
    ChooseNewThread -> do
      chooseThread
      setSchedulerAction ResumeCurrentThread
      ...
chooseThread = do
    r <- findM chooseThread' (reverse [minBound ..
    maxBound])
    when (r == Nothing) $ switchToIdleThread
chooseThread' prio = do
     q <- getQueue prio
     liftM isJust $ findM chooseThread'' q
chooseThread'' thread = do
     runnable <- isRunnable thread
     if not runnable then do
             tcbSchedDequeue thread
             return False
     else do
             switchToThread thread
             return True
```

used (ARMv6). These are explicit in the model, and we can therefore detect violations. *Foundationally* means that we do not just axiomatize the behavior of C on a high level, but we derive it from first principles as far as possible. For example, in our model of C, memory is a primitive function from addresses to bytes without type information or restrictions. On top of that, we specify how types like `unsigned int` are encoded, how structures are laid out, and how implicit and explicit type casts behave. We managed to lift this low-level memory model to a high-level calculus that allows efficient, abstract reasoning on the type-safe fragment of the kernel. We generate proof obligations assuring the safety of each pointer access and write. They state that the pointer in question must be non-null and of the correct alignment. They are typically easy to discharge. We generate similar obligations for all restrictions the C99 standard demands.

We treat a very large, pragmatic subset of C99 in the verification. It is a compromise between verification convenience and the hoops the kernel programmers were willing to jump through in writing their source. The following paragraphs describe what is *not* in this subset.

We do not allow the address-of operator & on local variables, because, for better automation, we make the assumption that local variables are separate from the heap. This could be violated if their address was available to pass on. It is the most far-reaching restriction we implement, because it is common in C to use local variable references for return parameters to avoid returning large types on the stack. We achieved compliance with this requirement by avoiding reference parameters as much as possible, and where they were needed, used pointers to global variables (which are not restricted).

One feature of C that is problematic for verification (and programmers) is the unspecified order of evaluation in expressions with side effects. To deal with this feature soundly, we limit how side effects can occur in expressions. If more than one function call occurs within an expression or the expression otherwise accesses global state, a proof obligation is generated to show that these functions are side-effect free. This proof obligation is discharged automatically.

We do not allow function calls through function pointers. (We do allow handing the address of a function to assembler code, e.g., for installing exception vector tables.) We also do not allow `goto` statements, or `switch` statements with fall-through cases. We support C99 compound literals, making it convenient to return structs from functions, and reducing the need for reference parameters. We do not allow compound literals to be *lvalues*. Some of these restrictions could be lifted easily, but the features were not required in seL4.

We did not use unions directly in seL4 and therefore do not support them in the verification (although that would be possible). Since the C implementation was derived from a functional program, all unions in seL4 are tagged, and many structs are packed bitfields. Like other kernel implementors, we do not trust GCC to compile and optimize bitfields predictably for kernel code. Instead, we wrote a small tool that takes a specification and generates C code with the necessary shifting and masking for such bitfields. The tool helps us to easily map structures to page table entries or other hardware-defined memory layouts. The generated code can be inlined and, after compilation on ARM, the result is more compact and faster than GCC's native bitfields. The tool not only generates the C code, it also automatically generates Isabelle/HOL specifications and proofs of correctness.

Figure 5. C code for part of the scheduler.

```
void setPriority(tcb_t *tptr, prio_t prio) {
  prio_t oldprio;
  if(thread_state_get_tcbQueued(tptr->tcbState)) {
    oldprio = tptr->tcbPriority;
    ksReadyQueues[oldprio] =
      tcbSchedDequeue(tptr, ksReadyQueues[oldprio]);
    if(isRunnable(tptr)) {
      ksReadyQueues[prio] =
        tcbSchedEnqueue(tptr, ksReadyQueues[prio]);
    }
    else {
      thread_state_ptr_set_tcbQueued(&tptr->tcbState,
                                     false);
    }
  }
  tptr->tcbPriority = prio;
}
```

Figure 5 shows part of the implementation of the scheduling functionality described in the previous sections. It is standard C99 code with pointers, arrays and structs. The `thread_state` functions used in Figure 5 are examples of generated bitfield accessors.

4.4. The proof

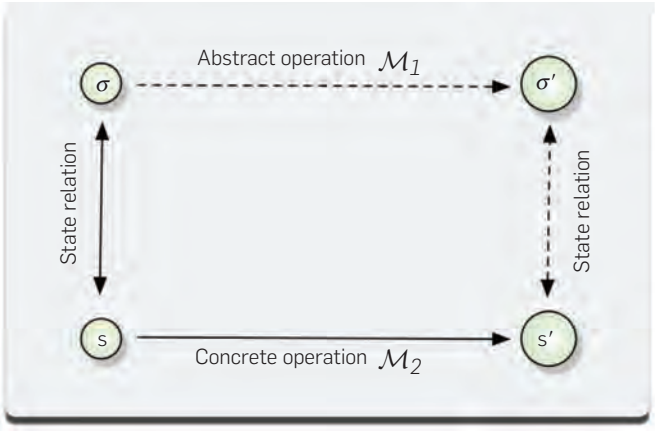
This section describes the main theorem we have shown and how its proof was constructed.

As mentioned, the main property we are interested in is functional correctness, which we prove by showing formal refinement. We have formalized this property for general state machines in Isabelle/HOL, and we instantiate each of the specifications in the previous sections into this state-machine framework.

We have also proved the well-known reduction of refinement to *forward simulation*, illustrated in Figure 6 where the solid arrows mean universal quantification and the dashed arrows existential: To show that a concrete state machine $\mathcal{M}_2$ refines an abstract one $\mathcal{M}_1$, it is sufficient to show that for each transition in $\mathcal{M}_2$ that may lead from an initial state s to a set of states s', there exists a corresponding transition on the abstract side from an abstract state σ to a set σ' (they are sets because the machines may be nondeterministic). The transitions *correspond* if there exists a relation R between the states s and σ such that for each concrete state in s' there is an abstract one in σ' that makes R hold between them again. This has to be shown for each transition with the same overall relation R. For externally visible state, we require R to be equality. For each refinement layer in Figure 2, we have strengthened and varied this proof technique slightly, but the general idea remains the same.

We now describe the instantiation of this framework to the seL4 kernel. We have the following types of transition in

Figure 6. Forward simulation.



our state machines: kernel transitions, user transitions, user events, idle transitions, and idle events. *Kernel transitions* are those that are described by each of the specification layers in increasing amount of detail. *User transitions* are specified as nondeterministically changing arbitrary user-accessible parts of the state space. *User-events* model kernel entry (trap instructions, faults, interrupts). *Idle transitions* model the behavior of the idle thread. Finally, *idle events* are interrupts occurring during idle time; other interrupts that occur during kernel execution are modelled explicitly and separately in each layer of Figure 2.

The model of the machine and the model of user programs remain the same across all refinement layers; only the details of kernel behavior and kernel data structures change. The fully nondeterministic model of the user means that our proof includes all possible user behaviors, be they benign, buggy, or malicious.

Let machine $\mathcal{M}_A$ denote the system framework instantiated with the abstract specification of Section 4.1, let machine $\mathcal{M}_E$ represent the framework instantiated with the executable specification of Section 4.2, and let machine $\mathcal{M}_C$ stand for the framework instantiated with the C program read into the theorem prover. Then we prove the following two, very simple-looking theorems:

THEOREM 1. $\mathcal{M}_E$ *refines* $\mathcal{M}_A$.

THEOREM 2. $\mathcal{M}_C$ *refines* $\mathcal{M}_E$.

Therefore, because refinement is transitive, we have

THEOREM 3. $\mathcal{M}_C$ *refines* $\mathcal{M}_A$.

4.5. Assumptions

Formal verification can never be absolute; it always must make fundamental assumptions. The assumptions we make are correctness of *the C compiler*, *the assembly code*, *the hardware*, *and kernel initialization*. We explain each of them in more detail below.

The *initialization code* takes up about 1.2 kloc of the kernel. The theorems in Section 4.4 only state correspondence between entry and exit points in each specification layer for a running kernel.

Assuming correctness of the *C compiler* means that we assume GCC correctly translates the seL4 source code in our C subset according to the ISO/IEC C99 standard,[4] that the formal model of our C subset accurately reflects this standard and that the model makes the correct architecture-specific assumptions for the ARMv6 architecture on the Freescale i.MX31 platform.

The assumptions on *hardware and assembly* mean that we do not prove correctness of the register save/restore and the potential context switch on kernel exit. Cache consistency, cache coloring, and TLB flushing requirements are part of the assembly implemented machine interface. These machine interface functions are called from C, and we assume they do not have any effect on the memory state of the C program. This is only true if they are used correctly.

The VM subsystem of seL4 is not assumed correct, but is treated differently from other parts of the proof. For our C semantics, we assume a traditional, flat view of in-kernel memory that is kept consistent by the kernel's VM subsystem. We make this consistency argument only informally; our model does not oblige us to prove it. We do however substantiate the model and informal argument by manually stated, machine-checked properties and invariants. This means we explicitly treat in-kernel VM in the proof, but this treatment is different from the high standards in the rest of our proof where we reason from first principles and the proof forces us to be complete.

This is the set of assumptions we picked. If they are too strong for a particular purpose, many of them can be eliminated combined with other research. For instance, we have verified the executable design of the boot code in an earlier design version. For context switching, Ni et al.[7] report verification success, and the Verisoft project[1] shows how to verify assembly code and hardware interaction. Leroy verified an optimizing C compiler[5] for the PowerPC and ARM architectures.

An often-raised concern is the question *What if there is a mistake in the proof?* The proof is machine-checked by Isabelle/HOL. So what if there is a bug in Isabelle/HOL? The proof checking component of Isabelle is small and can be isolated from the rest of the prover. It is extremely unlikely that there is a bug in this part of the system that applies in a correctness-critical way to our proof. If there was reason for concern, a completely independent proof checker could be written in a few hundred lines of code. Provers like Isabelle/HOL can achieve a degree of proof trustworthiness that far surpasses the confidence levels we rely on in engineering or mathematics for our daily survival.

5. EXPERIENCE AND LESSONS LEARNED

5.1. Verification effort

The project was conducted in three phases. First an initial kernel with limited functionality (no interrupts, single address space, and generic linear page table) was designed and implemented in Haskell, while the verification team mostly worked on the verification framework and generic proof libraries. In a second phase, the verification team developed the abstract spec and performed the

first refinement while the development team completed the design, Haskell prototype and C implementation. The third phase consisted of extending the first refinement step to the full kernel and performing the second refinement. The overall size of the proof, including framework, libraries, and generated proofs (not shown in the table) is 200,000 lines of Isabelle script.

Table 1 gives a breakdown for the effort and size of each of the layers and proofs. About 30 person months (pm) went into the abstract specification, Haskell prototype and C implementation (over all project phases), including design, documentation, coding, and testing.

This compares well with other efforts for developing a new microkernel from scratch: The Karlsruhe team reports that, on the back of their experience from building the earlier Hazelnut kernel, the development of the Pistachio kernel costs about 6 person years (py). SLOCCount with the "embedded" profile estimates the total cost of seL4 at 4 py. Hence, there is strong evidence that the detour via Haskell did not increase the cost, but was in fact a significant net cost saver.

The cost of the proof is higher, in total about 20 py. This includes significant research and about 9 py invested in formal language frameworks, proof tools, proof automation, theorem prover extensions, and libraries. The total effort for the seL4-specific proof was 11 py.

We expect that redoing a similar verification for a new kernel, using the same overall methodology, would reduce this figure to 6 py, for a total (kernel plus proof) of 8 py. This is only twice the SLOCCount estimate for a traditionally engineered system with no assurance.

The breakdown in Table 1 of effort between the two refinement stages is illuminating: almost 3:1. This is a reflection of the low-level nature of our Haskell prototype, which captures most of the properties of the final product. This is also reflected in the proof size—the first proof step contained most of the deep semantic content. 80% of the effort in the first refinement went into establishing invariants, only 20% into the actual correspondence proof. We consider this asymmetry a significant benefit, as the executable spec is more convenient and efficient to reason about than C.

The first refinement step led to some 300 changes in the abstract spec and 200 in the executable spec. About 50% of these changes relate to bugs in the associated algorithms or design. Examples are missing checks on user-supplied input, subtle side effects in the middle of an operation breaking global invariants, or over-strong assumptions about what is true during execution. The rest of the changes were introduced for verification convenience. The ability to change and rearrange code in discussion with the design team was an important factor in the verification team's productivity and was essential to complete the verification on time.

Table 1. Code and proof statistics.

	Haskell/C		Isabelle		Proof	
	pm	kloc	kloc	Invariants	py	klop
abst.	4	–	4.9	~75	8	110
exec.	24	5.7	13	~80		
impl.	2	8.7	15	0	3	55

The second refinement stage from executable spec to C uncovered 160 bugs, 16 of which were also found during testing, early application and static analysis. The bugs discovered in this stage were mainly typos, misreading the specification, or failing to update all relevant code parts for specification changes. Even though their cause was often simple, understandable human error, their effect in many cases was sufficient to crash the kernel or create security vulnerabilities. There were no deeper, algorithmic bugs in the C level, because the C code was written according to a very precise, low-level specification.

5.2. The cost of change

One issue of verification is the cost of proof maintenance: how much does it cost to reverify after changes are made to the kernel? This obviously depends on the nature of the change. We are not able to precisely quantify such costs, but our iterative verification approach has provided us with some relevant experience.

The best case is a *local, low-level code change*, typically an optimization that does not affect the observable behavior. We made such changes repeatedly, and found that the effort for reverification was always low and roughly proportional to the size of the change.

Adding new, *independent features*, which do not interact in a complex way with existing features, usually has a moderate impact. For example, adding a new system call to the seL4 API that atomically batches a specific, short sequence of existing system calls took one day to design and implement. Adjusting the proof took less than 1 person week.

Adding new, *large, cross-cutting features*, such as adding a complex new data structure to the kernel supporting new API calls that interact with other parts of the kernel, is significantly more expensive. We experienced such a case when progressing from the first to the final implementation, adding interrupts, ARM page tables, and address spaces. This change costs several pms to design and implement, and resulted in 1.5–2 py to reverify. It modified about 12% of existing Haskell code, added another 37%, and reverification cost about 32% of the time previously invested in verification. The new features required only minor adjustments of existing invariants, but lead to a considerable number of new invariants for the new code. These invariants had to be preserved over the whole kernel, not just the new features.

Unsurprisingly, *fundamental changes to existing features* are bad news. We experienced one such change when we added reply capabilities for efficient RPC as an API optimization after the first refinement was completed. Even though the code size of this change was small (less than 5% of the total code base), it violated key invariants about the way capabilities were used in the system until then and the amount of *conceptual* cross-cutting was huge. It took about 1 py or 17% of the original proof effort to reverify.

There is one class of otherwise frequent code changes

that does not occur after the kernel has been verified: implementation bug fixes.

6. CONCLUSION

We have presented our experience in formally verifying seL4. We have shown that full, rigorous, formal verification is practically achievable for OS microkernels.

The requirements of verification force the designers to think of the simplest and cleanest way of achieving their goals. We found repeatedly that this leads to overall better design, for instance, in the decisions aimed at simplifying concurrency-related verification issues.

Our future research agenda includes verification of the assembly parts of the kernel, a multi-core version of the kernel, as well as formal verification of overall system security and safety properties, including application components. The latter now becomes much more meaningful than previously possible: application proofs can rely on the abstract, formal kernel specification that seL4 is proven to implement.

Acknowledgments

We would like to acknowledge the contribution of the former team members on this verification project: Timothy Bourke, Jeremy Dawson, Jia Meng, Catherine Menon, and David Tsai.

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program. C

References

1. Alkassar, E., Schirmer, N., Starostin, A. Formal pervasive verification of a paging mechanism. *TACAS*. C.R. Ramakrishnan and J. Rehof, eds. Volume 4963 of *LNCS* (2008). Springer, 109–123.
2. Dennis, J.B., Van Horn, E.C. Programming semantics for multiprogrammed computations. *CACM 9* (1966), 143–155.
3. Elkaduwe, D., Klein, G., Elphinstone, K. Verified protection model of the seL4 microkernel. *VSTTE 2008—Verified Software: Theories, Tools & Experiments*. J. Woodcock and N. Shankar eds. Volume 5295 of *LNCS* (Toronto, Canada, Oct 2008), Springer, 99–114.
4. ISO/IEC. Programming languages—C. Technical Report 9899:TC2, ISO/IEC JTC1/SC22/WG14, May 2005.
5. Leroy, X. Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. *33rd POPL*. J.G. Morrisett and S.L.P. Jones, eds. (New York, NY, USA, 2006), ACM, 42–54.
6. Liedtke, J. Towards real microkernels. *CACM 39*, 9 (Sept 1996), 70–77.
7. Ni, Z., Yu, D., Shao. Z. Using XCAP to certify realistic system code: Machine context management. *20th TPHOLs*, volume 4732 of *LNCS* (Kaiserslautern, Germany, Sept 2007), Springer, 189–206.
8. Nipkow, T., Paulson, L., Wenzel, M. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*. Volume 2283 of *LNCS* (2002), Springer.
9. Ormandy, T., Tinnes, J. Linux null pointer dereference due to incorrect proto_ops initializations. http://www.cr0.org/misc/CVE-2009-2692.txt, 2009.
10. Saltzer, J.H., Schroeder, M.D. The protection of information in computer systems. *Proc. IEEE 63* (1975), 1278–1308.
11. Shapiro, J.S., Smith, J.M., Farber, D.J. EROS: A fast capability system. *17th SOSP* (Charleston, SC, USA, Dec 1999), 170–185.

Gerwin Klein National ICT Australia (NICTA), University of South Wales (UNSW), Sydney.

June Andronick NICTA, UNSW.

Kevin Elphinstone NICTA, UNSW.

Gernot Heiser NICTA, UNSW, Open Kernel Labs.

Dhammika Elkaduwe NICTA, UNSW, now at University of Peradeniya, Sri Lanka.

Kai Engelhardt NICTA, UNSW.

Rafal Kolanksi NICTA, UNSW.

Harvey Tuch NICTA, UNSW, now at VMWare.

Simon Winwood NICTA, UNSW.

David Cock NICTA.

Philip Derrin NICTA, now at Open Kernel Labs.

Michael Norrish NICTA, Australian National University, Canberra.

Thomas Sewell NICTA.

© 2010 ACM 0001-0782/10/0600 $10.00

Have you Considered All the Possibilities?

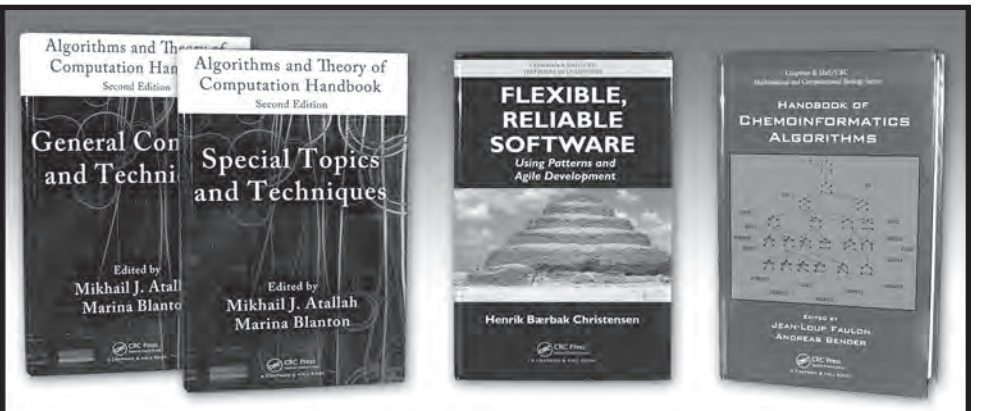
Save 20%
on these
New & Noteworthy Resources
from
Chapman & Hall/CRC Press

Save 20% on your online order by entering promo code **736DM** at checkout

Discount expires July 31, 2010

www.crcpress.com

CRC Press
Taylor & Francis Group
CHAPMAN & HALL BOOKS



New Edition of a Bestseller!

Algorithms and Theory of Computation Handbook, Second Edition—Two Volume Set

Praise for the Previous Edition

"... excellent survey of the state of the art ... highly recommended for anyone interested in algorithms, data structures and the theory of computation ... an indispensable book of reference ... ."
—R. Kemp, *Zentralblatt MATH, Vol. 926*

Catalog no. C8180
January 2010, 1938 pp.
ISBN: 978-1-58488-818-5
~~$179.95 / £114.00~~
$143.96 / £91.20

Flexible, Reliable Software
Using Patterns and Agile Development

"... brings together a careful selection of topics that are relevant, indeed crucial, for developing good quality software ... leads the reader through an experience of active learning."
—Michael Kölling, Bestselling Author and originator of the BlueJ and Greenfoot environments

Catalog no. C3622
May 2010, 527 pp.
ISBN: 978-1-4200-9362-9
~~$69.95 / £44.99~~
$55.96 / £35.99

Handbook of Chemoinformatics Algorithms

Providing an overview of the most common chemoinformatics algorithms, this book explains how to effectively apply algorithms and graph theory for solving chemical problems.

Catalog no. C2922
April 2010, 454 pp.
ISBN: 978-1-4200-8292-0
~~$99.95 / £63.99~~
$79.96 / £51.19