seL4: Formal Verification of an OS Kernel

Gerwin Klein[1,2], Kevin Elphinstone[1,2], Gernot Heiser[1,2,3]
June Andronick[1,2], David Cock[1], Philip Derrin[1*], Dhammika Elkaduwe[1,2‡], Kai Engelhardt[1,2]
Rafal Kolanski[1,2], Michael Norrish[1,4], Thomas Sewell[1], Harvey Tuch[1,2†], Simon Winwood[1,2]
[1] NICTA, [2] UNSW, [3] Open Kernel Labs, [4] ANU
ertos@nicta.com.au

ABSTRACT

Complete formal verification is the only known way to guarantee that a system is free of programming errors.

We present our experience in performing 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, and hardware, and we used a unique design approach that fuses formal and operating systems techniques. To our knowledge, this is the first formal proof of functional correctness of a complete, general-purpose operating-system kernel. Functional correctness means here that the implementation always strictly follows our high-level abstract specification of kernel behaviour. This encompasses traditional design and implementation safety properties such as the kernel will never crash, and it will never perform an unsafe operation. It also proves much more: we can predict precisely how the kernel will behave in every possible situation.

seL4, a third-generation microkernel of L4 provenance, comprises 8,700 lines of C code and 600 lines of assembler. Its performance is comparable to other high-performance L4 kernels.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability—*Verification*; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Verification, Design

Keywords

Isabelle/HOL, L4, microkernel, seL4

*Philip Derrin is now at Open Kernel Labs.
†Harvey Tuch is now at VMware.
‡Dhammika Elkaduwe is now at University of Peradeniya

To appear in the 22nd ACM Symposium on Principles of Operating Systems (SOSP), October 2009, Big Sky, MT, USA

1. INTRODUCTION

The security and reliability of a computer system can only be as good as that of the underlying operating system (OS) kernel. The kernel, defined as the part of the system executing in the most privileged mode of the processor, has unlimited hardware access. Therefore, any fault in the kernel's implementation has the potential to undermine the correct operation of the rest of the system.

General wisdom has it that bugs in any sizeable code base are inevitable. As a consequence, when security or reliability is paramount, the usual approach is to reduce the amount of privileged code, in order to minimise the exposure to bugs. This is a primary motivation behind security kernels and separation kernels [38, 54], the MILS approach [4], microkernels [1, 12, 35, 45, 57, 71] and isolation kernels [69], the use of small hypervisors as a minimal trust base [16, 26, 56, 59], as well as systems that require the use of type-safe languages for all code except some "dirty" core [7,23]. Similarly, the Common Criteria [66] at the strictest evaluation level requires the system under evaluation to have a "simple" design.

With truly small kernels it becomes possible to take security and robustness further, to the point where it is possible to *guarantee* the absence of bugs [22, 36, 56, 64]. This can be achieved by *formal, machine-checked verification*, providing mathematical proof that the kernel implementation is consistent with its specification and free from programmer-induced implementation defects.

We present seL4, a member of the L4 [46] microkernel family, designed to provide this ultimate degree of assurance of functional correctness by machine-assisted and machine-checked formal proof. We have shown the correctness of a very detailed, low-level design of seL4 and we have formally verified its C implementation. We assume the correctness of the compiler, assembly code, boot code, management of caches, and the hardware; we prove everything else.

Specifically, seL4 achieves the following:

- it is suitable for real-life use, and able to achieve performance that is comparable with the best-performing microkernels;
- its behaviour is precisely formally specified at an abstract level;
- its formal design is used to prove desirable properties, including termination and execution safety;
- its implementation is formally proven to satisfy the specification; and

- its access control mechanism is formally proven to provide strong security guarantees.

To our knowledge, seL4 is the first-ever general-purpose OS kernel that is fully formally verified for functional correctness. As such, it is a platform of unprecedented trustworthiness, which will allow the construction of highly secure and reliable systems on top.

The functional-correctness property we prove for seL4 is much 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 analyse specific aspects of the kernel, such as safe execution, but also provide a full specification and proof for the kernel's precise behaviour.

We have created a methodology for *rapid kernel design and implementation* that is a fusion of traditional operating systems and formal methods techniques. We found that our verification focus improved the design and was surprisingly often not in conflict with achieving performance.

In this paper, we present the design of seL4, discuss the methodologies we used, and provide an overview of the approach used in the formal verification from high-level specification to the C implementation. We also discuss the lessons we have learnt from the project, and the implications for similar efforts.

The remainder of this paper is structured as follows. In Sect. 2, we give an overview of seL4, of the design approach, and of the verification approach. In Sect. 3, we describe how to design a kernel for formal verification. In Sect. 4, we describe precisely what we verified, and identify the assumptions we make. In Sect. 5, we describe important lessons we learnt in this project, and in Sect. 6, we contrast our effort with related work.

2. OVERVIEW

2.1 seL4 programming model

This paper is primarily about the formal verification of the seL4 kernel, not its API design. We therefore provide only a brief overview of its main characteristics.

seL4 [20], similarly to projects at Johns Hopkins (Coyotos) and Dresden (Nova), is a third-generation microkernel, and is broadly based on L4 [46] and influenced by EROS [58]. It features abstractions for virtual address spaces, threads, inter-process communication (IPC), and, unlike most L4 kernels, capabilities for authorisation. Virtual address spaces have no kernel-defined structure; page faults are propagated via IPC to pager threads, responsible for defining the address space by mapping frames into the virtual space. Exceptions and non-native system calls are also propagated via IPC to support virtualisation. IPC uses synchronous and asynchronous *endpoints* (port-like destinations without in-kernel buffering) for inter-thread communication, with RPC facilitated via *reply capabilities*. Capabilities are segregated and stored in capability address spaces composed of capability container objects called *CNodes*.

As in traditional L4 kernels, seL4 device drivers run as normal user-mode applications that have access to device registers and memory either by mapping the device into the virtual address space, or by controlled access to device ports on Intel x86 hardware. seL4 provides a mechanism to receive notification of interrupts (via IPC) and acknowledge their receipt.

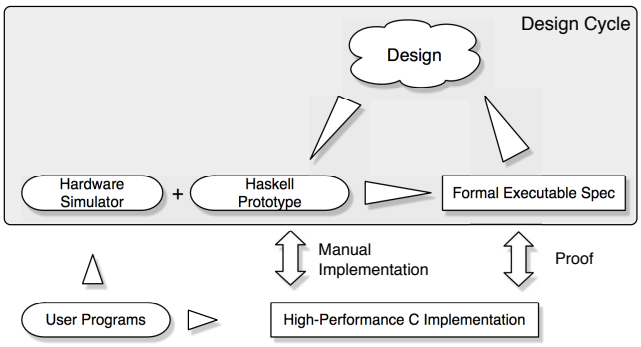

Figure 1: The seL4 design process

Memory management in seL4 is explicit: both in-kernel objects and virtual address spaces are protected and managed via capabilities. Physical memory is initially represented by *untyped* capabilities, which can be subdivided or *retyped* into kernel objects such as page tables, thread control blocks, CNodes, endpoints, and frames (for mapping in virtual address spaces). The model guarantees all memory allocation in the kernel is explicit and authorised.

Initial development of seL4, and all verification work, was done for an ARMv6-based platform, with a subsequent port of the kernel (so far without proof) to x86.

2.2 Kernel design process

OS developers tend to take a bottom-up approach to kernel design. High performance is obtained by managing the hardware efficiently, which leads to designs motivated by low-level details. In contrast, formal methods practitioners tend toward top-down design, as proof tractability is determined by system complexity. This leads to designs based on simple models with a high degree of abstraction from hardware.

As a compromise that blends both views, we adopted an approach [19, 22] based around an intermediate target that is readily accessible by both OS developers and formal methods practitioners. It uses the functional programming language Haskell to provide a programming language for OS developers, while at the same time providing an artefact that can be automatically translated into the theorem proving tool and reasoned about.

Fig. 1 shows our approach in more detail. The square boxes are formal artefacts that have a direct role in the proof. The double arrows represent implementation or proof effort, the single arrows represent design/implementation influence of artefacts on other artefacts. The central artefact is the Haskell prototype of the kernel. The prototype requires the design and implementation of algorithms that manage the low-level hardware details. To execute the Haskell prototype in a near-to-realistic setting, we link it with software (derived from QEMU) that simulates the hardware platform. Normal user-level execution is enabled by the simulator, while traps are passed to the kernel model which computes the result of the trap. The prototype modifies the user-level state of the simulator to appear as if a real kernel had executed in privileged mode.

This arrangement provides a prototyping environment that enables low-level design evaluation from both the user and kernel perspective, including low-level physical and virtual

memory management. It also provides a realistic execution environment that is binary-compatible with the real kernel. For example, we ran a subset of the Iguana embedded OS [37] on the simulator-Haskell combination. The alternative of producing the executable specification directly in the theorem prover would have meant a steep learning curve for the design team and a much less sophisticated tool chain for execution and simulation.

We restrict ourselves to a subset of Haskell that can be automatically translated into the language of the theorem prover we use. For instance, we do not make any substantial use of laziness, make only restricted use of type classes, and we prove that all functions terminate. The details of this subset are described elsewhere [19, 41].

While the Haskell prototype is an executable model and implementation of the final design, it is not the final production kernel. We *manually* re-implement the model in the C programming language for several reasons. Firstly, the Haskell runtime is a significant body of code (much bigger than our kernel) which would be hard to verify for correctness. Secondly, the Haskell runtime relies on garbage collection which is unsuitable for real-time environments. Incidentally, the same arguments apply to other systems based on type-safe languages, such as SPIN [7] and Singularity [23]. Additionally, using C enables optimisation of the low-level implementation for performance. While an automated translation from Haskell to C would have simplified verification, we would have lost most opportunities to micro-optimise the kernel, which is required for adequate microkernel performance.

2.3 Formal verification

The technique we use for formal verification is interactive, machine-assisted and machine-checked proof. Specifically, we use the theorem prover Isabelle/HOL [50]. Interactive theorem proving requires human intervention and creativity to construct and guide the proof. However, it has the advantage that it is not constrained to specific properties or finite, feasible state spaces, unlike more automated methods of verification such as static analysis or model checking.

The property we are proving is functional correctness in the strongest sense. Formally, we are showing *refinement* [18]: 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), 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. This is described elsewhere [11, 21], and we have not yet connected it to the proof presented here.

Fig. 2 shows the specification layers used in the verification of seL4; they are related by formal proof. Sect. 4 explains the proof and each of these layers in detail; here we give a short summary.

The top-most layer in the picture is the *abstract specification*: an operational model that is the main, complete specification of system behaviour. The abstract level contains enough detail to specify the outer interface of the kernel, e.g., how system-call arguments are encoded in binary form, and it describes in abstract logical terms the effect of each system call or what happens when an interrupt or VM fault occurs. It does not describe in detail how these effects are implemented in the kernel.

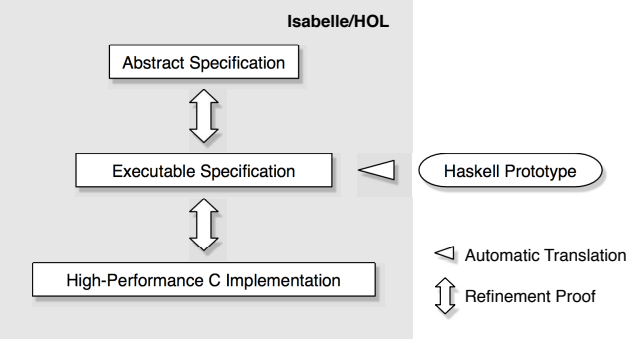

Figure 2: The refinement layers in the verification of seL4

The next layer down in Fig. 2 is the *executable specification* generated from Haskell into the theorem prover. The translation is not correctness-critical because we seek assurance about the generated definitions and ultimately C, not the Haskell source, which only serves as an intermediate prototype. The executable specification contains all data structure and implementation details we expect the final C implementation to have.

Finally, the bottom layer in this verification effort is the high-performance C implementation of seL4. For programs to be formally verified they must have formally-defined semantics. One of the achievements of this project is a very exact and faithful formal semantics for a large subset of the C programming language [62]. Even with a formal semantics of C in the logic of the theorem prover, we still have to read and translate the specific program into the prover. This is discussed in Sect. 4.3.

Verification can never be absolute; it always must make fundamental assumptions. In our work we stop at the source-code level, which implies that we assume at least the compiler and the hardware to be correct.

An often-raised concern is the question of proof correctness. More than 30 years of research in theorem proving has addressed this issue, and we can now achieve *a degree of trustworthiness of formal, machine-checked proof that far surpasses the confidence levels we rely on in engineering or mathematics for our daily survival.* We use two specific techniques: firstly, we work foundationally from first principles; mathematics, semantics, and Hoare logic are not axiomatised, but defined and proved. Secondly, the Isabelle theorem prover we are using can produce external proof representations that can be independently checked by a small, simple proof checker.

3. KERNEL DESIGN FOR VERIFICATION

The main body of the correctness proof can be thought of as showing Hoare triples on program statements and on functions in each of the specification levels. The proof in our refinement and Hoare logic framework decomposes along function boundaries. Each unit of proof has a set of pre-

conditions that need to hold prior to execution, a statement or sequence of statements in a function that modify the system state, and the post-conditions that must hold afterwards. The degree of difficulty in showing that pre- and post-conditions hold is directly related to the complexity of the statement, the state the statement can modify, and the complexity of the properties the pre- and post-conditions express. Around 80 % of the properties we show relate to preserving invariants.

To make verification of the kernel feasible, its design should minimise the complexity of these components. Ideally, the kernel code (and associated proofs) would consist of simple statements that rely on explicit local state, with simple invariants. These smaller elements could then be composed abstractly into larger elements that avoid exposing underlying local elements. Unfortunately, OS kernels are not usually structured like this, and generally feature highly inter-dependent subsystems [10].

As a consequence of our design goal of suitability for real-life use, our kernel design attempts to minimise the proof complexity without compromising performance. In this light we will now examine typical properties of kernels and discuss their effect on verification, including presenting specific features of our kernel design.

3.1 Global variables and side effects

Programming with global variables and with side effects is common in operating systems kernels and our verification technique has no problem dealing with them. However, implicit state updates and complex use of the same global for different purposes can make verification harder than necessary.

Global variables usually require stating and proving invariant properties. For example, if global scheduler queues are 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. 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 accidentally destroys the list or its properties. This proof can be easy or hard, depending on how modularly the global variable is used.

A hypothetical, problematic example would be a complex, overloaded page-table data structure that can represent translation, validity of memory regions, copy-on-write, zero-on-demand memory, and the location of data in swap space, combined with a relationship to the frame table. This would create a large, complex invariant for each of the involved data structures, and each of the involved operations would have to preserve all of it.

The treatment of globals becomes especially difficult if invariants are temporarily violated. For example, adding a new node to a doubly-linked list temporarily violates invariants that the list is well formed. Larger execution blocks of unrelated code, as in preemption or interrupts, should be avoided during that violation. We address these issues by limiting preemption points, and by deriving the code from Haskell, thus making side effects explicit and bringing them to the attention of the design team.

3.2 Kernel memory management

The seL4 kernel uses a model of memory allocation that exports control of the in-kernel allocation to appropriately authorised applications [20]. 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 of the kernel, which means we only need to prove that the mechanism works, not that the user-level policy makes sense. Obviously, moving it into userland does not change the fact that the memory-allocation module is part of the trusted computing base. It does mean, however, that such a module can be verified separately, and can rely on verified kernel properties.

The correctness of the allocation algorithm involves checks that new objects are wholly contained within an untyped (free) memory region and that they do not overlap with any other objects allocated from the region. Our memory allocation model keeps track of capability derivations in a tree-like structure, whose nodes are the capabilities themselves.

Before re-using 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, the 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.

3.3 Concurrency and non-determinism

Concurrency is the execution of computation in parallel (in the case of multiple hardware processors), or by non-deterministic interleaving via a concurrency abstraction like threads. Proofs about concurrent programs are *hard*, much harder than proofs about sequential programs.

While we have some ideas on how to construct verifiable systems on multiprocessors, they are outside the scope of this paper and left for future work. In this paper we focus on uniprocessor support where the degree of interleaving of execution and non-determinism can be controlled. However, even on a uniprocessor there is some remaining concurrency resulting from asynchronous I/O devices. seL4 avoids much of the complications resulting from I/O by running device drivers at user level, but it must still address interrupts.

Consider the small code fragment `A; X; B`, where `A` must establish the state that `X` relies on, `X` must establish the state `B` relies on, and so on. Concurrency issues in the verification of this code arise from yielding, interrupts and exceptions.

Yielding at `X` results in the potential execution of any reachable activity in the system. This implies `A` must establish the preconditions required for all reachable activities, and all reachable activities on return must establish the preconditions of `B`. Yielding increases complexity significantly and makes verification harder. Preemption is a non-deterministically optional yield. Blocking kernel primitives, such as in lock acquisition and waiting on condition variables, are also a form of non-deterministic yield.

By design, we side-step addressing the verification complexity of yield by using an event-based kernel execution model,

with a single kernel stack, and a mostly atomic application programming interface [25].

Interrupt complexity has two forms: non-deterministic execution of the interrupt handlers, and interrupt handling resulting in preemption (as a result of timer ticks). Theoretically, this complexity can be avoided by disabling interrupts during kernel execution. However, this would be at the expense of large or unbounded interrupt latency, which we consider unacceptable.

Instead, we run the kernel with interrupts mostly disabled, except for a small number of carefully-placed interrupt points. If, in the above code fragment, `X` is the interrupt point, `A` must establish the state that all interrupt handlers rely on, and all reachable interrupt handlers must establish or preserve the properties `B` relies on.

We simplify the problem further by implementing interrupt points via polling, rather than temporary enabling of interrupts. On detection of a pending interrupt, we explicitly return through the function call stack to the kernel/user boundary. At the boundary we leave a (potentially modified) event stored in the saved user-level registers. The interrupt becomes a new kernel event (prefixed to the pending user-triggered event). After the in-kernel component of interrupt handling, the interrupted event is restarted. This effectively re-tries the (modified) operation, including re-establishing all the preconditions for execution. In this way we avoid the need for any interrupt-point specific post-conditions for interrupt handlers, but still achieve Fluke-like partial preemptability [25].

The use of interrupt points creates a trade-off, controlled by the kernel designer, between proof complexity and interrupt processing latency. Almost all of seL4's operations have short and bounded latency, and can execute without any interrupt points at all. The exception is object destruction, whose cleanup operations are inherently unbounded, and are, of course, critical to kernel integrity.

We make these operations preemptable by storing the state of progress of destruction in the last capability referencing the object being destroyed; we refer to this as a *zombie* capability. This guarantees that the correctness of a restarted destroy is not dependent on user-accessible registers. Another advantage of this approach is that if another user thread attempts to destroy the zombie, it will simply continue where the first thread was preempted (a form of priority inheritance), instead of making the new thread dependent (blocked) on the completion of another.

Exceptions are similar to interrupts in their effect, but are synchronous in that they result directly from the code being executed and cannot be deferred. In the seL4 kernel we avoid exceptions completely and much of that avoidance is guaranteed as a side-effect of verification. Special care is required only for memory faults.

We avoid having to deal with virtual-memory exceptions in kernel code by mapping a fixed region of the virtual address space to physical memory, independent of whether it is actively used or not. The region contains all the memory the kernel can potentially use for its own internal data structures, and is guaranteed to never produce a fault. We prove that this region appears in every virtual address space.

Arguments passed to the kernel from user level are either transferred in registers or limited to preregistered physical frames accessed through the kernel region.

3.4 I/O

As described earlier we avoid most of the complexity of I/O by moving device drivers into protected user-mode components. When processing an interrupt event, our interrupt delivery mechanism determines the interrupt source, masks further interrupts from that specific source, notifies the registered user-level handler (device driver) of the interrupt, and unmasks the interrupt when the handler acknowledges the interrupt.

We coarsely model the hardware interrupt controller of the ARM platform to include interrupt support in the proof. The model includes existence of the controller, masking of interrupts, and that interrupts only occur if unmasked. This is sufficient to include interrupt controller access, and basic behaviour in the proof, without modelling correctness of the interrupt controller management in detail. The proof is set up such that it is easy to include more detail in the hardware model should it become necessary later to prove additional properties.

Our kernel contains a single device driver, the timer driver, which generates timer ticks for the scheduler. This is set up in the initialisation phase of the kernel as an automatically reloaded source of regular interrupts. It is not modified or accessed during the execution of the kernel. We did not need to model the timer explicitly in the proof, we just prove that system behaviour on each tick event is correct.

3.5 Observations

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, which tends to reduce the likelihood of bugs.

In a number of cases there were significant other benefits. This is particularly true for the design decisions aimed at simplifying concurrency-related verification issues. Non-preemptable execution (except for a few interrupt-points) has traditionally been used in L4 kernels to maximise average-case performance. Recent L4 kernels aimed at embedded use [32] have adopted an event-based design to reduce the kernel's memory footprint (due to the use of a single kernel stack rather than per-thread stacks).

4. seL4 VERIFICATION

This section describes each of the specification layers as well as the proof in more detail.

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 behaviour 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. In order to express these, we rarely make use of infinite types like natural numbers. Instead, we use finite machine words, such as 32-bit integers. We model memory and typed pointers explicitly. Otherwise, the data structures used in this abstract specification are high-level — essentially sets, lists, trees, functions and records. We make use of non-determinism in order to leave implementation choices to lower levels: If there are multiple correct results for an operation, this abstract layer

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

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

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 Fig. 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 non-deterministic 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 optimised 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 for instance 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 non-determinism 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.

Fig. 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 behaviour 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 optimisation 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 standard 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 used (ARMv6). These are explicit in the model, and we can therefore detect violations. *Foundationally* means that we do not just axiomatise the behaviour 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 [62, 63, 65]. 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.

```
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
```

Figure 4: Haskell code for `schedule`.

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 to use local variable references for return parameters of large types that we do not want to pass 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).

```
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: C code for part of the scheduler.

```
configureTimer :: irq => unit machine_m
resetTimer :: unit machine_m
setCurrentPD :: paddr => unit machine_m
setHardwareASID :: hw_asid => unit machine_m
invalidateTLB :: unit machine_m
invalidateHWASID :: hw_asid => unit machine_m
invalidateMVA :: word => unit machine_m
cleanCacheMVA :: word => unit machine_m
cleanCacheRange :: word => word => unit machine_m
cleanCache :: unit machine_m
invalidateCacheRange :: word => word => unit machine_m
getIFSR :: word machine_m
getDFSR :: word machine_m
getFAR :: word machine_m
getActiveIRQ :: (irq option) machine_m
maskInterrupt :: bool => irq => unit machine_m
```

Figure 6: Machine interface functions.

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 depends on global state, a proof obligation is generated to show that these functions are side-effect free. This proof obligation is discharged automatically by Isabelle.

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 optimise 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 [13].

Fig. 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 Fig. 5 are examples of generated bitfield accessors.

4.4 Machine model

Programming in C is not sufficient for implementing a kernel. There are places where the programmer has to go outside the semantics of C to manipulate hardware directly. In the easiest case, this is achieved by writing to memory-mapped device registers, as for instance with a timer chip; in other cases one has to drop down to assembly to implement the required behaviour, as for instance with TLB flushes.

Presently, we do not model the effects of certain direct hardware instructions because they are too far below the abstraction layer of C. Of these, cache and TLB flushes are relevant for the correctness of the code, and we rely on traditional testing for these limited number of cases. Higher assurance can be obtained by adding more detail to the machine model—we have phrased the machine interface such that future proofs about the TLB and cache can be added with minimal changes. Additionally, required behaviour can be guaranteed by targeted assertions (e.g., that page-table updates always flush the TLB), which would result in further proof obligations.

The basis of this formal model of the machine is the internal state of the relevant devices, collected in one record `machine_state`. For devices that we model more closely, such as the interrupt controller, the relevant part in `machine_state` contains details such as which interrupts are currently masked. For the parts that we do not model, such as the TLB, we leave the corresponding type unspecified, so it can be replaced with more details later.

Fig. 6 shows our machine interface. The functions are all of type `X machine_m` which restricts any side effects to the `machine_state` component of the system. Most of the functions return nothing (type `unit`), but change the state of a device. In the abstract and executable specification, these functions are implemented with maximal non-determinism. This means that in the extreme case they may arbitrarily change their part of the machine state. Even for devices that we model, we are careful to leave as much behaviour as possible non-deterministic. The less behaviour we prescribe, the less assumptions the model makes about the hardware.

In the seL4 implementation, the functions in Fig. 6 are implemented in C where possible, and otherwise in assembly; we must check (but we do not prove) that the implementations match the assumptions we make in the levels above. An example is the function `getIFSR`, which on ARM returns the instruction fault status register after a page fault. For this function, which is basically a single assembly instruction, we only assume that it does not change the memory state of the machine, which is easy to check.

4.5 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 formalised 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 Fig. 7: 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 non-deterministic). 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 each refinement layer in Fig. 2, we have strengthened and varied this proof technique slightly, but the general idea remains the same. Details are published elsewhere [14, 70].

We now describe the instantiation of this framework to the seL4 kernel. We have the following types of transition in 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 non-deterministically changing arbitrary user-accessible parts of the state space. *User events* model kernel entry (trap instructions, faults, interrupts). *Idle transitions* model the behaviour 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 Fig. 2.

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

Let machine $\mathcal{M}_A$ denote the system framework instantiated with the abstract specification of Sect. 4.1, let machine $\mathcal{M}_E$ represent the framework instantiated with the executable specification of Sect. 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 $\mathcal{M}_A$.*

Assumptions. The assumptions we make are correctness of the C compiler, the assembly code level, and the hardware. We currently omit correctness of the boot/initialisation code which takes up about 1.2 kLOC of the kernel; the theorems above state correspondence between the kernel entry and exit points in each specification layer. We describe these assumptions in more detail below and discuss their implications.

For the C level, we assume that the GCC compiler correctly implements our subset according to the ISO/IEC C99 standard [39], that the formal model of our C subset accurately reflects this standard and that it makes the correct architecture-specific assumptions for the ARMv6 architecture on the Freescale i.MX31 platform.

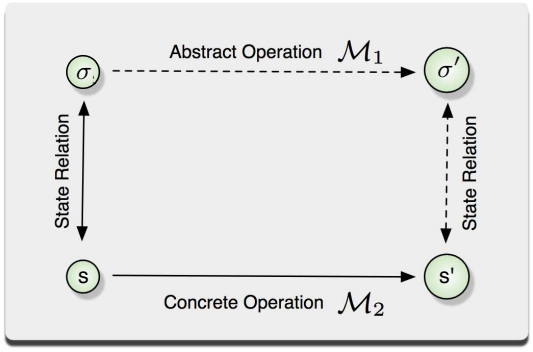

Figure 7: Forward Simulation.

The assumptions on the hardware and assembly level mean that we do not prove correctness of the register save/restore and the potential context switch on kernel exit. As described in Sect. 4.4, cache consistency, cache colouring, 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 under the assumption they are used correctly.

In-kernel memory and code access is translated by the TLB on ARM processors. For our C semantics, we assume a traditional, flat view of in-kernel memory that is consistent because all kernel reads and writes are performed through a constant one-to-one VM window which the kernel establishes in every address space. We make this consistency argument only informally; our model does not oblige us to prove it. We do however substantiate the model by manually stated properties and invariants. This means our treatment of in-kernel virtual memory is different to the high standards in the rest of our proof where we reason from first principles and the proof forces us to be complete.

These are not fundamental limitations of the approach, but a decision taken to achieve the maximum outcome with available resources. For instance, we have verified the executable design of the boot code in an earlier design version. For context switching, Ni et al. [49] report verification success, and the Verisoft project [3] showed how to verify assembly code and hardware interaction. Leroy verified an optimising C compiler [44] for the PowerPC architecture. We have also shown that kernel VM access and faults can be modelled foundationally from first principles [42].

Assurance. Having outlined the limitations of our verification, we now discuss the properties that are proved.

Overall, we show that the behaviour of the C implementation is fully captured by the abstract specification. This is a strong statement, as it allows us to conduct all further analysis of properties that can be expressed as Hoare triples on the massively simpler abstract specification instead of a complex C program. Coverage is complete. Any remaining implementation errors (deviations from the specification) can only occur below the level of C.

A cynic might say that an implementation proof only shows that the implementation has precisely the same bugs that the specification contains. This is true: the proof does not guarantee that the specification describes the behaviour the user expects. The difference is the degree of abstraction and the absence of whole classes of bugs. In the same notation, the abstract specification is one third the size of the C code and works with concepts that are simpler and faster to reason about. The current level of abstraction is low enough to be precise for the operational behaviour of the kernel. To analyse specific properties of the system, one might also introduce another, even higher level of abstraction that contains only the aspects relevant for the property. An example is our access control model of seL4 [11, 21].

In addition to the implementation correctness statement, our strengthened proof technique for forward simulation [14] implies that both $\mathcal{M}_E$ and $\mathcal{M}_C$ never fail and always have defined behaviour. This means the kernel can never crash or otherwise behave unexpectedly as long as our assumptions hold. This includes that all assertions[1] in the kernel design are true on all code paths, and that the kernel never accesses a null pointer or a misaligned pointer.

We proved that all kernel API calls terminate and return to user level. There is no possible situation in which the kernel can enter an infinite loop. Since the interface from user level to the abstract specification is binary compatible with the final implementation, our refinement theorem implies that the kernel does all argument checking correctly and that it can not be subverted by buggy encodings, spurious calls, maliciously constructed arguments to system calls, buffer overflow attacks or other such vectors from user level. All these properties hold with the full assurance of machine-checked proof.

As part of the refinement proof between levels $\mathcal{M}_A$ and $\mathcal{M}_E$, we had to show a large number of invariants. These invariants are not merely a proof device, but provide valuable information and assurance in themselves. In essence, they collect information about what we know to be true of each data structure in the kernel, before and after each system call, and also for large parts during kernel execution where some of these invariants may be temporarily violated and re-established later. The overall proof effort was clearly dominated by invariant proofs, with the actual refinement statements between abstract and executable specification accounting for at most 20 % of the total effort for that stage. There is not enough space in this paper to enumerate all the invariant statements we have proved, but we will attempt a rough categorisation, show a few representatives, and give a general flavour.

There are four main categories of invariants in our proof: low-level memory invariants, typing invariants, data structure invariants, and algorithmic invariants.

The first two categories could in part be covered by a type-safe language: low-level memory invariants include that there is no object at address 0, that kernel objects are aligned to their size, and that they do not overlap. The typing invariants say that each kernel object has a well-defined type and that its references in turn point to objects of the right type. An example would be a capability slot containing a reference to a thread control block (TCB). The invariant would state that the type of the first object is a capability-table entry and that its reference points to a valid object in memory with type TCB. Intuitively, this invariant implies that *all reachable, potentially used references in the kernel—be it in capabilities, kernel objects or other data structures—always point to an object of the expected type.* This is a necessary condition for safe execution: we need to know that pointers point to well-defined and well-structured data, not garbage. This is also a dynamic property, because objects can be deleted and memory can be re-typed at runtime. Note that the main invariant is about *potentially used* references. We do allow some references, such as in ARM page table objects, to be temporarily left dangling, as long as we can prove that these dangling references will never be touched. Our typing invariants are stronger than those one would expect from a standard programming language type system. They are context dependent and they include value ranges such as using only a certain number of bits for hardware address space identifiers (ASIDs). They also often exclude specific values such as -1 or 0 as valid values because these are used in C to indicate success or failure of the corresponding operations. Typing invariants are usually simple to state and for large parts of the code, their preservation can be proved automatically. There are only two operations where the proof is difficult: removing and retyping objects. Type preservation for these two operations is the main reason for a large number of other kernel invariants.

The third category of invariants are classical data structure invariants like correct back links in doubly-linked lists, a statement that there are no loops in specific pointer structures, that other lists are always terminated correctly with NULL, or that data structure layout assumptions are interpreted the same way everywhere in the code. These invariants are not especially hard to state, but they are frequently violated over short stretches of code and then re-established later — usually when lists are updated or elements are removed.

The fourth and last category of invariants that we identify in our proof are algorithmic invariants that are specific to how the seL4 kernel works. These are the most complex invariants in our proof and they are where most of the proof effort was spent. These invariants are either required to prove that specific optimisations are allowed (e.g. that a check can be left out because the condition can be shown to be always true), or they are required to show that an operation executes safely and does not violate other invariants, especially not the typing invariant. Examples of simple algorithmic invariants are that the idle thread is always in thread state *idle*, and that only the idle thread is in this state. Another one is that the global kernel memory containing kernel code and data is mapped in all address spaces. Slightly more involved are relationships between the existence of capabilities and thread states. For instance, if a Reply capability exists to a thread, this thread must always be waiting to receive a reply. This is a non-local property connecting the existence of an object somewhere in memory with a particular state of another object somewhere else. Other invariants formally describe a general symmetry principle that seL4 follows: if an object x has a reference to another object y, then there is a reference in object y that can be used to find object x directly or indirectly. This fact is exploited heavily in the delete operation to clean up all remaining references to an

[1] One might think that assertions are pointless in a verified kernel. In fact, they are not only a great help during development, they also convey a useful message from the kernel designer to the verifier about an important invariant in the code, and as such aid verification.

object before it is deleted.

The reason this delete operation is safe is complicated. Here is a simplified, high-level view of the chain of invariants that show an efficient local pointer test is enough to ensure that deletion is globally safe:

(1) If an object is live (contains references to other objects), there exists a capability to it somewhere in memory. (2) If an untyped capability c_1 covers a sub-region of another capability c_2, then c_1 must be a descendant of c_2 according to the capability derivation tree (CDT). (3) If a capability c_1 points to a kernel object whose memory is covered by an untyped capability c_2, then c_1 must be a descendant of c_2.

With these, we have: If an untyped capability has no children in the CDT (a simple pointer comparison according to additional data structure invariants), then all kernel objects in its region must be non-live (otherwise there would be capabilities to them, which in turn would have to be children of the untyped capability). If the objects are not live and no capabilities to them exist, there is no further reference in the whole system that could be made unsafe by the type change because otherwise the symmetry principle on references would be violated. Deleting the object will therefore preserve the basic typing and safety properties. Of course we also have to show that deleting the object preserves all the new invariants we just used as well.

We have proved over 150 invariants on the different specification levels, most are interrelated, many are complex. All these invariants are expressed as formulae on the kernel state and are proved to be preserved over all possible kernel executions.

5. EXPERIENCE AND LESSONS LEARNT

5.1 Performance

IPC performance is the most critical metric for evaluation in a microkernel in which all interaction occurs via IPC. We have evaluated the performance of seL4 by comparing IPC performance with L4, which has a long history of data points to draw upon [47].

Publicly available performance for the Intel XScale PXA 255 (ARMv5) is 151 in-kernel cycles for a one-way IPC [43], and our experiments with OKL4 2.1 [51] on the platform we are using (Freescale i.MX31 evaluation board based on a 532 MHz ARM1136JF-S which is an ARMv6 ISA) produced 206 cycles as a point for comparison—this number was produced using a hand-crafted assembly-language path. The non-optimised C version in the same kernel took 756 cycles. These are hot-cache measurements obtained using the processor cycle counter.

Our measurement for seL4 is 224 cycles for one-way IPC in an optimised C path, which is approaching the performance of optimised assembly-language IPC paths for other L4 kernels on ARM processors. This puts seL4 performance into the vicinity of the fastest L4 kernels.

At the time of writing, this optimised IPC C path is not yet in the verified code base, but it is within the verifiable fragment of C and is not fundamentally different from the rest of the code.

5.2 Verification effort

The overall code statistics are presented in Table 1.

	Haskell/C LOC	Isabelle LOC	Invariants	Proof LOP
abst.	—	4,900	~ 75	110,000
exec.	5,700	13,000	~ 80	55,000
impl.	8,700	15,000	0	

Table 1: Code and proof statistics.

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.

The abstract spec took about 4 person months (pm) to develop. About 2 person years (py) went into the Haskell prototype (over all project phases), including design, documentation, coding, and testing. The executable spec only required setting up the translator; this took 3 pm.

The initial C translation was done in 3 weeks, in total the C implementation took about 2 pm, for a total cost of 2.2 py including the Haskell effort.

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 cost about 6 py [17]. SLOCCount [68] 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. This means that our development process can be highly recommended even for projects not considering formal verification.

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 re-doing 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. It certainly compares favourable to industry rules-of-thumb of \$10k/LOC for Common Criteria EAL6 certification, which would be \$87M for seL4, yet provides far less assurance than formal verification.

The breakdown of effort between the two refinement stages is illuminating: The first refinement step (from abstract to executable spec) consumed 8 py, the second (to concrete spec) less than 3 py, almost a 3:1 breakdown. This is a reflection of the low-level nature of our Haskell implementation, 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 the C level.

Our formal refinement framework for C made it possible to avoid proving any invariants on the C code, speeding up this stage of the proof significantly. We proved only few additional invariants on the executable spec layer to substantiate optimisations in C.

The first refinement step lead 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, the rest were introduced for verification convenience. The ability to change and rearrange code in discussion with the design team (to predict performance impact) was an important factor in the verification team's productivity. It is a clear benefit of the approach described in Sect. 2.2 and was essential to complete the verification in the available time.

By the time the second refinement started, the kernel had been used by a number of internal student projects and the x86 port was underway. Those two activities uncovered 16 defects in the implementation before verification had started in earnest, the formal verification has uncovered another 144 defects and resulted in 54 further changes to the code to aid in the proof. None of the bugs found in the C verification stage were deep in the sense that the corresponding algorithm was flawed. This is because the C code was written according to a very precise, low-level specification which was already verified in the first refinement stage. Algorithmic bugs found in the first stage were mainly 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 bugs discovered in the second proof from executable spec to C were mainly typos, misreading the specification, or failing to update all relevant code parts for specification changes. Simple typos also made up a surprisingly large fraction of discovered bugs in the relatively well tested executable specification in the first refinement proof, which suggests that normal testing may not only miss hard and subtle bugs, but also a larger number of simple, obvious faults than one may expect. 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. Other more interesting bugs found during the C implementation proof were missing exception case checking, and different interpretations of default values in the code. For example, the interrupt controller on ARM returns 0xFF to signal that no interrupt is active which is used correctly in most parts of the code, but in one place the check was against NULL instead.

The C verification also lead to changes in the executable and abstract specifications: 44 of these were to make the proof easier; 34 were implementation restrictions, such as the maximum size of virtual address space identifiers, which the specifications should make visible to the user.

5.3 The cost of change

An obvious issue of verification is the cost of proof maintenance: how much does it cost to re-verify after changes made to the kernel? It obviously depends on the nature of the change, specifically the amount of code it changes, the number of invariants it affects, and how localised it is. We are not able to quantify such costs, but our iterative approach to verification has provided us with some relevant experience.

The best case are *local, low-level code changes*, typically optimisations that do not affect the observable behaviour. We made such changes repeatedly, and found that the effort for re-verification 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 effect. 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 pw.

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 cost several pms to design and implement, and resulted in 1.5–2 py to re-verify. It modified about 12 % of existing Haskell code, added another 37 %, and re-verification 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 have to be preserved over the whole kernel API, not just the new features.

Unsurprisingly, *fundamental changes to existing features* are bad news. We had one example of such a change when we added reply capabilities for efficient RPC as an API optimisation after the first refinement was completed. Reply capabilities are created on the fly in the receiver of an IPC and are treated in most cases like other capabilities. They are single-use, and thus deleted immediately after use. This fundamentally broke a number of properties and invariants on capabilities. Creation and deletion of capabilities require a large number of preconditions to execute safely. The operations were carefully constrained in the kernel. Doing them on the fly required complex preconditions to be proved for many new code paths. Some of these turned out not to be true, which required extra work on special-case proofs or changes to existing invariants (which then needed to be re-proved for the whole kernel). Even though the code size of this change was small (less than 5 % of the total code base), the comparative amount of *conceptual* cross-cutting was huge. It took about 1 py or 17 % of the original proof effort to re-verify.

There is one class of otherwise frequent code changes that does not occur after the kernel has been verified: implementation bug fixes.

6. RELATED WORK

We briefly summarise the literature on OS verification. Klein [40] provides a comprehensive overview.

The first serious attempts to verify an OS kernel were in the late 1970s UCLA Secure Unix [67] and the Provably Secure Operating System (PSOS) [24]. Our approach mirrors the UCLA effort in using refinement and defining functional correctness as the main property to prove. The UCLA project managed to finish 90 % of their specification and 20 % of their proofs in 5 py. They concluded that invariant reasoning dominated the proof effort, which we found confirmed in our project.

PSOS was mainly focussed on formal kernel design and never completed any substantial implementation proofs. Its design methodology was later used for the Kernelized Secure Operating System (KSOS) [52] by Ford Aerospace. The

Secure Ada Target (SAT) [30] and the Logical Coprocessor Kernel (LOCK) [55] are also inspired by the PSOS design and methodology.

In the 1970s, machine support for theorem proving was rudimentary. Basic language concepts like pointers still posed large problems. The UCLA effort reports that the simplifications required to make verification feasible made the kernel an order of magnitude slower [67]. We have demonstrated that with modern tools and techniques, this is no longer the case.

The first real, completed implementation proofs, although for a highly idealised OS kernel are reported for KIT, consisting of 320 lines of artificial, but realistic assembly instructions [8].

Bevier and Smith later produced a formalisation of the Mach microkernel [9] without implementations proofs. Other formal modelling and proofs for OS kernels that did not proceed to the implementation level include the EROS kernel [57], the high-level analysis of SELinux [5, 29] based on FLASK [60], and the MASK [48] project which was geared towards information-flow properties.

The VFiasco project [36] and later the Robin project [61] attempted to verify C++ kernel implementations. They managed to create a precise model of a large, relevant part of C++, but did not verify substantial parts of a kernel.

Heitmeyer et al [33] report on the verification and Common Criteria certification of a "software-based embedded device" featuring a small (3,000 LOC) separation kernel. They show data separation only, not functional correctness. Although they seem to at least model the implementation level, they did not conduct a machine-checked proof directly on the C-code.

Hardin et al [31] formally verified information-flow properties of the AAMP7 microprocessor [53], which implements the functionality of a static separation kernel in hardware. The functionality provided is less complex than a general purpose microkernel—the processor does not support online reconfiguration of separation domains. The proof goes down to a low-level design that is in close correspondence to the micro code. This correspondence is not proven formally, but by manual inspection.

A similar property was recently shown for Green Hills' Integrity kernel [28] during a Common Criteria EAL6+ certification [27]. The Separation Kernel Protection Profile [38] of Common Criteria shows data separation only. It is a weaker property than full functional correctness.

A closely related contemporary project is Verisoft [2], which is attempting to verify not only the OS, but a whole software stack from verified hardware up to verified application programs. This includes a formally verified, non-optimising compiler for their own Pascal-like implementation language. Even if not all proofs are completed yet, the project has successfully demonstrated that such a verification stack for full functional correctness can be achieved. They have also shown that verification of assembly-level code is feasible. However, Verisoft accepts two orders of magnitude slow-down for their highly-simplified VAMOS kernel (e.g. only single-level page tables) and that their verified hardware platform VAMP is not widely deployed. We deal with real C and standard tool chains on ARMv6, and have aimed for a commercially deployable, realistic microkernel.

Other formal techniques for increasing the trustworthiness of operating systems include static analysis, model checking and shape analysis. Static analysis can in the best case only show the absence of certain classes of defects such as buffer overruns. Model checking in the OS space includes SLAM [6] and BLAST [34]. They can show specific safety properties of C programs automatically, such as correct API usage in device drivers. The terminator tool [15] increases reliability of device drivers by attempting to prove termination automatically. Full functional correctness of a realistic microkernel is still beyond the scope of these automatic techniques.

Implementations of kernels in type-safe languages such as SPIN [7] and Singularity [23] offer increased reliability, but they have to rely on traditional "dirty" code to implement their language runtime, which tends to be substantially bigger than the complete seL4 kernel. While type safety is a good property to have, it is not very strong. The kernel may still misbehave or attempt, for instance, a null pointer access. Instead of randomly crashing, it will report a controlled exception. In our proof, we show a variant of type safety for the seL4 code. Even though the kernel deliberately breaks the C type system, it only does so in a safe way. Additionally, we prove much more: that there will never be any such null pointer accesses, that the kernel will never crash and that the code always behaves in strict accordance with the abstract specification.

7. CONCLUSIONS

We have presented our experience in formally verifying seL4. We have shown that full, rigorous, formal verification is practically achievable for OS microkernels with very reasonable effort compared to traditional development methods.

Although we have not invested significant effort into optimisation, we have shown that optimisations are possible and that performance does not need to be sacrificed for verification. The seL4 kernel is practical, usable, and directly deployable, running on ARMv6 and x86.

Collateral benefits of the verification include our rapid prototyping methodology for kernel design. We observed a confluence of design principles from the formal methods and the OS side, leading to design decisions such as an event-based kernel that is mostly non-preemptable and uses interrupt polling. These decisions made the kernel design simpler and easier to verify without sacrificing performance. Evidence suggests that taking the detour via a Haskell prototype increased our productivity even without considering verification.

Future work in this project includes verification of the assembly parts of the kernel, a multi-core version of the kernel, as well as application verification. 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.

Compared to the state of the art in software certification, the ultimate degree of trustworthiness we have achieved is redefining the standard of highest assurance.

Acknowledgements

We thank Timothy Bourke, Timothy Roscoe, and Adam Wiggins for valued feedback on drafts of this article. We also would like to acknowledge the contribution of the former team members on this verification project: Jeremy Dawson, Jia Meng, Catherine Menon, and David Tsai.

NICTA is funded by the Australian Government as repre-

sented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

8. REFERENCES

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *1986 Summer USENIX*, pages 93–112, 1986.

[2] E. Alkassar, M. Hillebrand, D. Leinenbach, N. Schirmer, A. Starostin, and A. Tsyban. Balancing the load — leveraging a semantics stack for systems verification. *JAR*, 42(2–4), 2009.

[3] E. Alkassar, N. Schirmer, and A. Starostin. Formal pervasive verification of a paging mechanism. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Alg. for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *LNCS*, pages 109–123. Springer, 2008.

[4] J. Alves-Foss, P. W. Oman, C. Taylor, and S. Harrison. The MILS architecture for high-assurance embedded systems. *Int. J. Emb. Syst.*, 2:239–247, 2006.

[5] M. Archer, E. Leonard, and M. Pradella. Analyzing security-enhanced Linux policy specifications. In *POLICY '03: Proc. 4th IEEE Int. WS on Policies for Distributed Systems and Networks*, pages 158–169. IEEE Computer Society, 2003.

[6] T. Ball and S. K. Rajamani. SLIC: A specification language for interface checking. Technical Report MSR-TR-2001-21, Microsoft Research, 2001.

[7] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *15th SOSP*, Dec 1995.

[8] W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.

[9] W. R. Bevier and L. Smith. A mathematical model of the Mach kernel: Atomic actions and locks. Technical Report 89, Computational Logic Inc., Apr 1993.

[10] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: its extracted software architecture. In *ICSE '99: Proc. 21st Int. Conf. on Software Engineering*, pages 555–563. ACM, 1999.

[11] A. Boyton. A verified shared capability model. In G. Klein, R. Huuck, and B. Schlich, editors, *4th WS Syst. Softw. Verification SSV'09*, ENTCS, pages 99–116. Elsevier, Jun 2009.

[12] P. Brinch Hansen. The nucleus of a multiprogramming operating system. *CACM*, 13:238–250, 1970.

[13] D. Cock. Bitfields and tagged unions in C: Verification through automatic generation. In B. Beckert and G. Klein, editors, *VERIFY'08*, volume 372 of *CEUR Workshop Proceedings*, pages 44–55, Aug 2008.

[14] D. Cock, G. Klein, and T. Sewell. Secure microkernels, state monads and scalable refinement. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *21st TPHOLs*, volume 5170 of *LNCS*, pages 167–182. Springer, Aug 2008.

[15] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. Proving that programs eventually do something good. In *34th POPL*. ACM, 2007.

[16] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *16th SOSP*, pages 351–366, Oct 2007.

[17] U. Dannowski. Personal communication.

[18] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Number 47 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.

[19] P. Derrin, K. Elphinstone, G. Klein, D. Cock, and M. M. T. Chakravarty. Running the manual: An approach to high-assurance microkernel development. In *ACM SIGPLAN Haskell WS*, Sep 2006.

[20] D. Elkaduwe, P. Derrin, and K. Elphinstone. Kernel design for isolation and assurance of physical memory. In *1st IIES*, pages 35–40. ACM SIGOPS, Apr 2008.

[21] D. Elkaduwe, G. Klein, and K. Elphinstone. Verified protection model of the seL4 microkernel. In J. Woodcock and N. Shankar, editors, *VSTTE 2008 — Verified Softw.: Theories, Tools & Experiments*, volume 5295 of *LNCS*, pages 99–114. Springer, Oct 2008.

[22] K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser. Towards a practical, verified kernel. In *11th HotOS*, pages 117–122, May 2007.

[23] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *1st EuroSys Conf.*, pages 177–190, Apr 2006.

[24] R. J. Feiertag and P. G. Neumann. The foundations of a provably secure operating system (PSOS). In *AFIPS Conf. Proc., 1979 National Comp. Conf.*, Jun 1979.

[25] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann. Interface and execution models in the Fluke kernel. In *3rd OSDI*. USENIX, Feb 1999.

[26] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *19th SOSP*, Oct 2003.

[27] Green Hills Software, Inc. INTEGRITY-178B separation kernel security target version 1.0. http://www.niap-ccevs.org/cc-scheme/st/st_vid10119-st.pdf, 2008.

[28] Greenhills Software, Inc. Integrity real-time operating system. http://www.ghs.com/products/rtos/integrity.html, 2008.

[29] J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka. Verifying information flow goals in security-enhanced Linux. *Journal of Computer Security*, 13(1):115–134, 2005.

[30] J. T. Haigh and W. D. Young. Extending the noninterference version of MLS for SAT. *IEEE Trans. on Software Engineering*, 13(2):141–150, 1987.

[31] D. S. Hardin, E. W. Smith, and W. D. Young. A robust machine code proof framework for highly secure applications. In *ACL2'06: Proc. Int. WS on the ACL2 theorem prover and its applications*. ACM, 2006.

[32] G. Heiser. Hypervisors for consumer electronics. In *6th IEEE CCNC*, 2009.

[33] C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. McLean. Formal specification and verification of data separation in a separation kernel for an embedded

system. In *CCS '06: Proc. 13th Conf. on Computer and Communications Security*, pages 346–355. ACM, 2006.

[34] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In *SPIN'03, Workshop on Model Checking Software*, 2003.

[35] M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro. Reducing TCB size by using untrusted components — small kernels versus virtual-machine monitors. In *11th SIGOPS Eur. WS*, Sep 2004.

[36] M. Hohmuth and H. Tews. The VFiasco approach for a verified operating system. In *2nd PLOS*, Jul 2005.

[37] Iguana. http://www.ertos.nicta.com.au/software/kenge/iguana-project/latest/.

[38] Information Assurance Directorate. *U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness*, Jun 2007. Version 1.03. http://www.niap-ccevs.org/cc-scheme/pp/pp.cfm/id/pp_skpp_hr_v1.03/.

[39] ISO/IEC. Programming languages — C. Technical Report 9899:TC2, ISO/IEC JTC1/SC22/WG14, May 2005.

[40] G. Klein. Operating system verification — an overview. *Sādhanā*, 34(1):27–69, Feb 2009.

[41] G. Klein, P. Derrin, and K. Elphinstone. Experience report: seL4 — formally verifying a high-performance microkernel. In *14th ICFP*, Aug 2009.

[42] R. Kolanski and G. Klein. Types, maps and separation logic. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Proc. TPHOLs'09*, volume 5674 of *LNCS*. Springer, 2009.

[43] L4HQ. http://l4hq.org/arch/arm/.

[44] X. Leroy. Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In J. G. Morrisett and S. L. P. Jones, editors, *33rd POPL*, pages 42–54. ACM, 2006.

[45] J. Liedtke. Improving IPC by kernel design. In *14th SOSP*, pages 175–188, Dec 1993.

[46] J. Liedtke. Towards real microkernels. *CACM*, 39(9):70–77, Sep 1996.

[47] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, and T. Jaeger. Achieved IPC performance (still the foundation for extensibility). In *6th HotOS*, pages 28–31, May 1997.

[48] W. B. Martin, P. White, A. Goldberg, and F. S. Taylor. Formal construction of the mathematically analyzed separation kernel. In *ASE '00: Proc. 15th IEEE Int. Conf. on Automated software engineering*, pages 133–141. IEEE Computer Society, 2000.

[49] Z. Ni, D. Yu, and Z. Shao. Using XCAP to certify realistic system code: Machine context management. In *Proc. TPHOLs'07*, volume 4732 of *LNCS*, pages 189–206. Springer, Sep 2007.

[50] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[51] OKL4 web site. http://okl4.org.

[52] T. Perrine, J. Codd, and B. Hardy. An overview of the kernelized secure operating system (KSOS). In *Proceedings of the Seventh DoD/NBS Computer Security Initiative Conference*, pages 146–160, Sep 1984.

[53] Rockwell Collins, Inc. *AAMP7r1 Reference Manual*, 2003.

[54] J. M. Rushby. Design and verification of secure systems. In *8th SOSP*, pages 12–21, 1981.

[55] O. Saydjari, J. Beckman, and J. Leaman. Locking computers securely. In *10th National Computer Security Conference*, pages 129–141, Sep 1987.

[56] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *16th SOSP*, pages 335–350, Oct 2007.

[57] J. S. Shapiro, D. F. Faber, and J. M. Smith. State caching in the EROS kernel—implementing efficient orthogonal peristence in a pure capability system. In *5th IWOOOS*, pages 89–100, Nov 1996.

[58] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A fast capability system. In *17th SOSP*, Dec 1999.

[59] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: Three case studies. In *1st EuroSys Conf.*, pages 161–174, Apr 2006.

[60] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *8th USENIX Security Symp.*, Aug 1999.

[61] H. Tews, T. Weber, and M. Völp. A formal model of memory peculiarities for the verification of low-level operating-system code. In R. Huuck, G. Klein, and B. Schlich, editors, *Proc. 3rd Int. WS on Systems Software Verification (SSV'08)*, volume 217 of *ENTCS*, pages 79–96. Elsevier, Feb 2008.

[62] H. Tuch. *Formal Memory Models for Verifying C Systems Code*. PhD thesis, UNSW, Aug 2008.

[63] H. Tuch. Formal verification of C systems code: Structured types, separation logic and theorem proving. *JAR*, 42(2–4):125–187, 2009.

[64] H. Tuch, G. Klein, and G. Heiser. OS verification — now! In *10th HotOS*, pages 7–12. USENIX, Jun 2005.

[65] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *34th POPL*, pages 97–108, Jan 2007.

[66] US National Institute of Standards. *Common Criteria for IT Security Evaluation*, 1999. ISO Standard 15408. http://csrc.nist.gov/cc/.

[67] B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verification of the UCLA Unix security kernel. *CACM*, 23(2):118–131, 1980.

[68] D. A. Wheeler. SLOCCount. http://www.dwheeler.com/sloccount/, 2001.

[69] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. In *5th OSDI*, Dec 2002.

[70] S. Winwood, G. Klein, T. Sewell, J. Andronick, D. Cock, and M. Norrish. Mind the gap: A verification framework for low-level C. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Proc. TPHOLs'09*, volume 5674. Springer, 2009.

[71] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *CACM*, 17:337–345, 1974.