Comprehensive Formal Verification of an OS Microkernel

GERWIN KLEIN, JUNE ANDRONICK, KEVIN ELPHINSTONE, TOBY MURRAY, THOMAS SEWELL, RAFAL KOLANSKI, and GERNOT HEISER, NICTA and UNSW, Sydney, Australia

We present an in-depth coverage of the comprehensive machine-checked formal verification of seL4, a general-purpose operating system microkernel.

We discuss the kernel design we used to make its verification tractable. We then describe the functional correctness proof of the kernel's C implementation and we cover further steps that transform this result into a comprehensive formal verification of the kernel: a formally verified IPC fastpath, a proof that the binary code of the kernel correctly implements the C semantics, a proof of correct access-control enforcement, a proof of information-flow noninterference, a sound worst-case execution time analysis of the binary, and an automatic initialiser for user-level systems that connects kernel-level access-control enforcement with reasoning about system behaviour. We summarise these results and show how they integrate to form a coherent overall analysis, backed by machine-checked, end-to-end theorems.

The seL4 microkernel is currently not just the only general-purpose operating system kernel that is fully formally verified to this degree. It is also the only example of formal proof of this scale that is kept current as the requirements, design and implementation of the system evolve over almost a decade. We report on our experience in maintaining this evolving formally verified code base.

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

General Terms: Verification, Security, Reliability

Additional Key Words and Phrases: seL4, Isabelle/HOL, operating systems, microkernel, L4

ACM Reference Format:
Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive formal verification of an OS microkernel. ACM Trans. Comput. Syst. 32, 1, Article 2 (February 2014), 70 pages.
DOI: http://dx.doi.org/10.1145/2560537

1. INTRODUCTION

This article presents a detailed coverage of the comprehensive formal verification of the seL4 microkernel, from its initial functional correctness proof to more recent results, which extend the assurance argument up to higher-level security properties and down to the binary level of its implementation.

The target of our verification, the kernel, is the most critical part of a system, which is our motivation for starting system verification with this component. The

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.
Authors' address: NICTA, 223 Anzac Pde, UNSW Sydney NSW 2052, Australia; Correspondence email: gernot@nicta.com.au.

DOI: http://dx.doi.org/10.1145/2560537

customary definition of a kernel is 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. Microkernels are motivated by the desire to minimise the exposure to bugs by reducing the amount of privileged code [Brinch Hansen 1970; Wulf et al. 1974; Accetta et al. 1986; Liedtke 1993; Shapiro et al. 1996; Hohmuth et al. 2004]. This is also the primary motivation behind separation kernels [Rushby 1981; Information Assurance Directorate 2007], the MILS approach [Alves-Foss et al. 2006], isolation kernels [Whitaker et al. 2002], the use of small hypervisors as a minimal trust base [Garfinkel et al. 2003; Singaravelu et al. 2006; Seshadri et al. 2007; Criswell et al. 2007], as well as systems that require the use of type-safe languages for all code except some "dirty" core [Bershad et al. 1995; Fähndrich et al. 2006].

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 [Tuch et al. 2005; Hohmuth and Tews 2005; Seshadri et al. 2007; Elphinstone et al. 2007] and to establish the presence of high-level security properties [Klein et al. 2011; Sewell et al. 2011; Murray et al. 2013; Sewell et al. 2013]. This can be achieved by *formal, machine-checked verification*, providing mathematical proof that firstly the kernel implementation is consistent with its specification and free from programmer-induced implementation defects, and secondly that the specification satisfies desirable high-level properties that carry through to the code and binary level.

The seL4 microkernel is a member of the L4 microkernel family [Liedtke 1996], designed for providing provably strong security mechanisms while retaining the high performance that is customary in the L4 family and considered essential for real-world use. The radical size reduction in microkernels 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, our work shows that with modern techniques and careful design, an operating system (OS) microkernel is entirely within the realm of full formal verification—not just for functional correctness, but also for a full range of further formal analysis.

Most of the key results summarised here have appeared previously in separate publications [Klein et al. 2009b; Sewell et al. 2011; Blackham et al. 2011; Murray et al. 2013; Sewell et al. 2013; Boyton et al. 2013]. The main contribution of this article is to give an overall picture and an in-depth coverage of the entire, comprehensive verification of seL4. We tie the individual results together, and extensively analyse our experience in sustained long-term verification.

The central piece of this work is still the initial functional correctness verification of seL4 [Klein et al. 2009b] in the theorem prover Isabelle/HOL [Nipkow et al. 2002]. This property is stronger and more precise than what automated techniques such as model checking, static analysis or kernel implementations in type-safe languages can achieve. It not only analyses specific aspects of the kernel, such as safe execution, but also provides a full specification and proof for the kernel's precise behaviour down to its C implementation on the ARM platform.

The following additional results, summarised here, extend this proof in the strongest sense. They directly compose with the formal functional correctness statement and yield end-to-end theorems about the C source code semantics in the theorem prover Isabelle/HOL:

—a functional correctness proof for a high-performance, hand-optimised inter-process communication (IPC) fastpath;
—a proof of correct access-control enforcement, in particular integrity and authority confinement [Sewell et al. 2011];

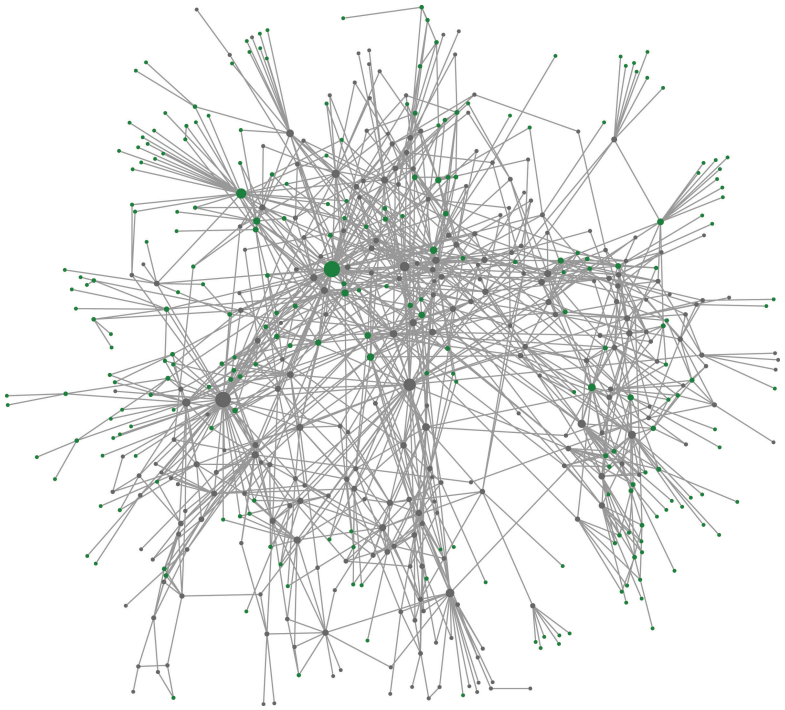


Fig. 1. Call graph of the seL4 microkernel. Vertices represent functions, and edges invocations.

—a proof of information-flow noninterference, which shows that seL4 can be configured as a separation kernel to provide strong isolation [Murray et al. 2013];
—a proof of user-level system initialisation that connects to the access-control model of the two security proofs above [Boyton et al. 2013].

The last three of these proofs reduce the gap between requirements as understood by humans, and the formal functional specification: they more directly and obviously describe properties we want this kernel to have. As we will discuss in the experience section, these additional proofs connect to appropriately abstract functional specification layers of seL4, and can thus be conducted with significantly less effort than the initial functional correctness proof.

Two further results on seL4 strengthen the assurance argument we can give on the binary level of the kernel:

—an automatic proof of refinement between the semantics of the kernel binary after compilation/linking and the C source code semantics used in the functional correctness proof [Sewell et al. 2013];
—an automatic static analysis of the seL4 binary to provide a sound worst-case execution time (WCET) profile of all system calls [Blackham et al. 2011].

To achieve a higher degree of automation, both of these analyses leave the logical framework of Isabelle/HOL. While the WCET analysis is a separate tool with its own highly detailed hardware model, the binary correctness verification stays logically compatible, using the Cambridge ARM semantics [Fox 2003; Fox and Myreen 2010] in the HOL4 prover [Slind and Norrish 2008], connecting directly to the Isabelle/HOL C semantics of the functional correctness proof. Since it hands off proof obligations to

automatic SMT solvers such as Z3 [de Moura and Bjørner 2008], the resulting theorem is strictly speaking not an end-to-end theorem in Isabelle/HOL. Even so, the result is still a refinement statement that composes logically with the other functional correctness results.

With this, the functional correctness and security proofs do not need to assume the correctness of compiler and linker any more. We still assume correctness of hand-written assembly code, boot code, management of caches, and the hardware; we prove everything else. Information-flow noninterference comes with its own set of additional hardware assumptions that we detail in Section 5.4.

This verification makes seL4 the only kernel analysed to the degree of assurance and detail presented here, and still, to our knowledge, the only general-purpose OS kernel that is fully formally verified for functional correctness with machine-checked end-to-end theorems. It is also the only evolving formally-verified code base of the order of 10 000 lines of code and we report on maintaining it for almost a decade together with its now 480 000 lines of Isabelle proofs and specifications.

The remainder of this article is structured as follows. In Section 2, we give an overview of seL4, of the design approach, and of the verification approach. In Section 3, we describe how to design a kernel for formal verification. Section 4 summarises the extended functional correctness verification and identifies the assumptions we make there. Section 5 does the same for the security theorems, and Section 6 discusses how to build trustworthy systems on top of seL4, including worst-case execution time and correct user-level initialisation of systems. In Section 7, we describe lessons we learnt in this work and in maintaining a code base with increasing large-scale formal proofs over an extended period of time.

2. OVERVIEW

Before diving into the details of how the kernel has been designed and verified, this section describes the kernel's main features and the general design and verification processes and artefacts. It is largely based on our previous article [Klein et al. 2009b], updated in the light of recent progress.

2.1. seL4 Programming Model

In this section, we provide an overview of seL4's main characteristics in order to provide sufficient background for later discussion. We postpone our examination of verification-related design issues to Section 3. Nonverification related API design issues are examined in a related publication [Elphinstone and Heiser 2013]. Detailed documentation and a seL4 binary are available for download [NICTA 2013a].

seL4 is a third-generation microkernel, loosely similar to Coyotos [2008] and Nova [Steinberg and Kauer 2010]). It is broadly based on L4 [Liedtke 1996] and influenced by EROS [Shapiro et al. 1999]. Like L4, it features abstractions for virtual address spaces, threads, IPC, and, unlike most earlier L4 kernels, an explicit in-kernel memory management model and capabilities for authorisation.

Authority in seL4 is conferred by possession of a capability [Dennis and Van Horn 1966]. Capabilities are segregated and stored in capability address spaces composed of capability container objects called *CNodes*. seL4 has six system calls, of which five require possession of a capability (the sixth is a *yield* to the scheduler). The five system calls are IPC primitives that are used either to invoke services implemented by other processes (using capabilities to port-like *endpoints* that facilitate message passing), or invoke kernel services (using capabilities to kernel objects, such as a thread control block (TCB)). While the number of system calls is small, the kernel's effective API is the sum of all the interfaces presented by all kernel object types.

Kernel Memory Management in seL4 is explicit [Elkaduwe et al. 2008; Elkaduwe 2010]. All kernel data structures are either statically allocated at boot time, or are dynamically allocated first-class objects in the API. Kernel objects thus act as both in-kernel data structures, and user-invoked fine-grained kernel services. Kernel objects include TCBs, CNodes, and level-1 and level-2 page tables (termed *PageDirectories* and *PageTables*).

Authority over free memory is encapsulated in an *untyped memory* object. Creating new kernel objects explicitly involves invoking the *retype* method of an untyped memory object, which allocates the memory for the object, initialises it, and returns a capability to the new object. We discuss kernel memory management further in Section 3.3.

Virtual Address Spaces are formed by explicit manipulation of virtual-memory-related kernel objects: PageDirectories, PageTables, ASIDPools[1] and Frames (mappable physical memory). As such, address spaces have no kernel-defined structure (except for a protected region reserved for the seL4 kernel itself). Whether the user-level system is Exokernel like, a multiserver, or a para-virtualised monolithic OS is determined by user-level via a *map* and *unmap* interface to Frames and PageTables. The distribution of authority to the kernel virtual memory (VM) objects ultimately determines the scope of influence over virtual and physical memory.

Threads are the active entity in seL4. By associating a CNode and a virtual address space with a thread, user-level policies create high-level abstractions, such as processes or virtual machines.

IPC is supported in two forms: synchronous message passing via *endpoints* (port-like destinations without in-kernel buffering), and asynchronous notification via *asynchronous endpoints* (rendezvous objects consisting of a single in-kernel word that is used to combine IPC sends using a logical *or*). Remote procedure call semantics are facilitated over synchronous IPC via *reply capabilities*. Send capabilities are *minted* from an initial endpoint capability. Send capabilities feature an immutable *badge* which is used by the specific endpoint's IPC recipients to distinguish which send capability (and thus which authority) was used to send a message. The unforgeable badge, represented by an integer value, is delivered with the message.

Exceptions are propagated via synchronous IPC to each thread's exception handler (registered in the TCB via a capability to an endpoint). Similarly, page faults are also propagated using synchronous IPC to a thread's page fault handler. Non-native system calls are treated as exceptions to support virtualisation.

Device Drivers run as 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 x86 hardware. seL4 provides a mechanism to receive notification of interrupts (via asynchronous IPC) and acknowledge their receipt.

The seL4 kernel runs on ARM and x86 platforms. Verified versions currently exist for ARMv6 and ARMv7. The port of seL4 to the Intel x86 platform, including support for VT-d and VT-x extensions, is currently unverified. In addition, there is an experimental x86 multicore version of seL4, together with a formal argument that lifts large parts of the functional correctness proof presented here to the multicore version [von Tessin 2013].

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

[1]Address-space identifiers (ASIDs) are a user-visible software artefact of the current implementation, and not directly related to hardware ASIDs used in tagged TLBs. They are not fundamental in any way. We expect to remove them in later kernel design iterations.

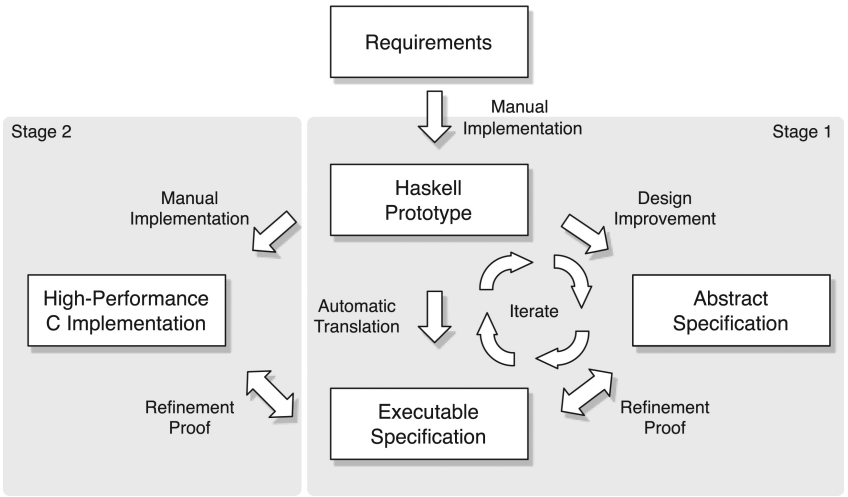


Fig. 2. The seL4 design process.

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 based around an intermediate target that is readily accessible to both OS developers and formal methods practitioners [Elphinstone et al. 2007; Derrin et al. 2006]. 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.

Figure 2 shows our approach in more detail. The central artefact is the Haskell prototype of the kernel. The prototype is derived from the (informal) requirements and embodies the design and implementation of algorithms that manage the low-level hardware details. It can be translated automatically into the theorem prover Isabelle/HOL to form an executable, design-level specification of the kernel. The abstract, high-level, functional specification of the kernel can be developed manually, concurrently and semi-independently, giving further feedback into the design process.

Already in very early development stages it is possible to link the Haskell prototype with hardware simulators such as QEMU. In this link, normal user-level execution is enabled by the simulator, while traps are passed to the kernel model which computes the result of the trap. This enables the developers to run and test user-level binary programs against early kernel API versions. The 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 [NICTA 2006] on the simulator-Haskell combination. The alternative of producing the executable specification directly in the theorem prover would have meant a steeper 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 [Derrin et al. 2006; Klein et al. 2009a].

While the Haskell prototype is an executable model and an implementation of the final design, it is not the final production kernel. As shown in Figure 2, we manually re-implemented the model in the C programming language. We did this for several reasons. First, the Haskell runtime is a significant body of code (much bigger than our kernel) which would be hard to verify for correctness. Second, 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 [Bershad et al. 1995] and Singularity [Fähndrich et al. 2006]. 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 we consider necessary for adequate microkernel performance.

After design and prototyping led to stabilised versions of executable and abstract specification, verification between these could commence, and the C code could be implemented concurrently. Eventually, when the process finishes, both specifications and C code are linked by proof. The original Haskell prototype is not part of the proof chain.

2.3. Formal Verification

The technique we use for formal verification is almost exclusively interactive, machine-assisted and machine-checked proof. Specifically, we use the theorem prover Isabelle/HOL [Nipkow et al. 2002]. 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 only two aspects of the overall verification that use different techniques are the worst-case execution time analysis, which is an automated static analysis of the binary, and the translation validation step between C semantics and the binary of the kernel, which is a combination of interactive theorem proving, and two standard SMT solvers as well as an automated tool that coordinates these provers.

In previous work [Klein et al. 2009b, 2010] we had concentrated on the functional correctness of seL4, which we later extended with binary verification [Sewell et al. 2013] and the functional correctness verification of seL4's IPC fastpath described below in Section 4.6.

These results together constitute a functional correctness property in the strongest sense. Formally, we are showing *refinement* [de Roever and Engelhardt 1998]: 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 gives us an end-to-end theorem in Isabelle/HOL that the same property holds for the kernel source code. The binary verification establishes another refinement theorem, this time between the source code and the binary, albeit in a separate tool.

In general, a functional correctness proof shows that a system is implemented correctly. It does not show that the right system was implemented, that is, that the kernel has the high-level properties that are needed. While this latter verification can never be fully discharged by formal proof, because the mental concept of "the right system" is not a formal one, we can still use formal verification to provide strong evidence that the system has at least some of the desired properties.

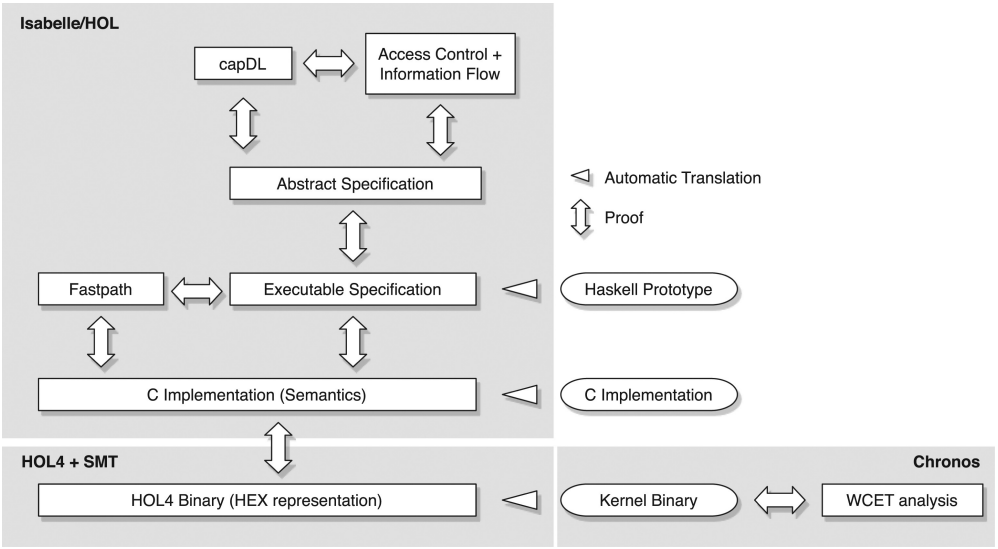


Fig. 3. The proof layers in the verification of seL4.

This was the goal of a second stream of work on seL4 after the initial functional correctness verification. The results of this stream appear as two additional specification layers in Figure 3. The difference to the other specification layers is that the new layers are no longer sufficient for describing the full functional behaviour of the kernel. Instead, they abstract away from a large amount of detail to be able to formally phrase intuitive high-level properties.

The two layers are a language for configuring seL4-based systems (*capDL*), and a collection of security properties and proofs about seL4.

The *capDL* layer is described in detail in Section 6.1. It supports developers in building trustworthy systems on top of seL4, and can be used to provably achieve a desired system configuration [Boyton et al. 2013]. CapDL stands for capability distribution language [Kuz et al. 2010]. The language describes the detailed protection state of a system snapshot in terms of kernel capabilities, at a level of detail sufficient to initialise the system into this configuration. It has a human-readable textual representation, an XML representation for system dumps and processing, a graphical representation for visual system inspection, and a formal representation for reasoning about detailed access-control states.

The second new layer, *Access Control + Information Flow* on the top right in Figure 3, represents high-level security properties we have proved about seL4. The classic security properties that the access-control mechanisms of any OS microkernel should provide are availability, authority confinement, integrity, and confidentiality. Availability means that an unauthorised application should not be able to deny service in terms of the resources the kernel manages, that is, processor time and memory resources. Authority confinement in a dynamic capability system means that authority cannot be escalated or transferred to another entity without explicit authorisation. Integrity means that an application cannot write to or change resource state without authorisation, and the dual, confidentiality, means no unauthorised read operations can be performed [Bishop 2003].

Section 5 describes proofs for all of these properties, and their respective security policy formalisations. In particular, we report on the explicit authority confinement

and integrity proof that has appeared previously [Sewell et al. 2011], and on a proof of intransitive noninterference [Murray et al. 2012, 2013] which includes confidentiality. Together, these results imply availability, because confidentiality and integrity together give isolation and isolated components cannot deny each others' resources. Availability also has a temporal aspect that is not covered by the integrity and noninterference results. It is instead covered by the worst-case execution time analysis described in Section 6.2.

There is a formal connection between a capDL description of a system and its higher-level security and information-flow policies. This connection allows us to connect system initialisation with known security policy states. We describe this connection in Section 6.1.

The rest of Figure 3 shows the specification layers used in the functional verification of seL4; like the other layers they are related by formal proof. Section 4 explains the proof and each of these layers in detail; here we give a short summary.

The topmost layer beneath capDL and security 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, for example, 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 virtual memory fault occurs. It does not describe in detail how these effects are implemented in the kernel.

The next layer down in Figure 3 is the *executable specification* generated from Haskell by automated import into the theorem prover. The translation is not correctness-critical because we seek assurance about the generated definitions and ultimately C and the binary, 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.

Next to the executable specification, on the left in Figure 3, is the IPC fastpath. Its verification followed a slightly different approach than the rest of the kernel: there exists an executable, detailed specification of the fastpath, but no abstract-level specification. Instead, after proving that the C fastpath implements its executable specification, we additionally prove that this specification is equivalent to the existing normal IPC path in the executable specification. This is the natural correctness criterion for an optimisation: it should provide precisely the same behaviour as the unoptimised code, just faster.

The next layer down in the functional verification 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 [Tuch 2008]. 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 Section 4.3.

Finally, the bottom layer in Figure 3 is the binary ELF file of the kernel as it is loaded onto the machine. This binary file contains the compiled and linked code of the kernel. As with the C and Haskell levels, the formal proof tools do not reason directly about the binary itself, but about its representation, in this case in the HOL4 [Slind and Norrish 2008] theorem prover. For the binary, this representation is extremely simple: a string of hexadecimal numbers as stored in the file. The meaning of these numbers is given by the Cambridge ARM semantics [Fox 2003], which has been extensively validated against real hardware [Fox and Myreen 2010]. As mentioned previously, the proof between C semantics and binary semantics of the kernel is not done exclusively in Isabelle, but also uses the HOL4 prover as well as two SMT solvers. Section 4.7 describes this part of the verification in detail.

The final verification aspect of seL4 we present in this article is the worst-case execution time analysis of the kernel binary, on the bottom right in Figure 3. While the rest of the verification stack reasons mostly about the functional and security behaviour of the kernel, this analysis covers the most important nonfunctional property a high-assurance system should have: precise timing predictions. This analysis is conducted as a standalone static analysis with an accurate (and sound) processor model, including pipelining and other performance-relevant features. We describe this verification in Section 6.2.

Verification can never be absolute; it must always make fundamental assumptions. In this work, we verify the kernel with high-level security properties down to the binary level, but we still assume correctness of TLB and cache flushing operations as well as the correctness of machine interface functions implemented in handwritten assembly. Of course, we also assume hardware correctness. Section 4.8 gives details on the precise assumptions of this verification and how they can be reduced even further.

It is worth noting that even though the proofs summarised previously have been conducted in multiple separate research projects over a time of more than 8 years in total, the results reported here are all fully integrated with each other and formally connected to the same version of the kernel (except the worst-case execution-time analysis in Section 6.2, which uses different techniques). Proofs and kernel code have evolved during this time and all related proof and code artefacts have been maintained to stay synchronised. This would have been infeasible without mechanised proof and fully automated proof checking.

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: first, we work foundationally from first principles; mathematics, semantics, and Hoare logic are not axiomatised, but defined and proved. Second, the Isabelle theorem prover we are using is an LCF-style prover [Gordon et al. 1979] where soundness critical code is concentrated in a relatively small proof kernel. Additionally, it can produce external proof representations that can be independently checked by a small, simple proof checker [Berghofer 2003].

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 preconditions 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 in the design verification 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 interdependent subsystems [Bowman et al. 1999].

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 OS kernels and our verification technique has no problem dealing with them. However, implicit state updates and complex use of the same global state for different purposes can make verification harder than necessary.

Global variables (or more generally, globally reachable data structures) 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 structure 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 view of the data structure, and each of the involved operations would have to preserve all aspects of that invariant.

The treatment of global data structures 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 pre-emption or interrupts, should be avoided during that violation. We address these issues by limiting pre-emption points (see Section 3.4 for further discussion), and by deriving the code from Haskell, thus making side-effects explicit and bringing them to the attention of the design team.

The top-level global state in seL4 is small. It consists of five arrays, two integers, and four pointers. It facilitates scheduling (including the current thread pointer), IRQ managment, ASID management, and a single physical frame globally mapped across all address spaces. All other state is reachable via capabilities or page tables associated with a thread (i.e., via the current-thread pointer).

While only anecdotal, we found that prototyping via Haskell reduced the tendency for side-effect based programming as it required more effort on the part of the programmer to explicitly make global state visible. We repeatedly observed that new kernel programmers experimenting in C would create side-effects and unintentionally violate invariants due to the lack of any barrier to state reachable from the current thread pointer. This was less of an issue for experienced and disciplined programmers.

3.2. Kernel Phases

The majority of seL4's API calls are divided (by convention) into two phases: a checking phase and an execute phase. The checking phase validates any arguments and confirms the authority to execute the call, and thus establishes the preconditions required for execution. The execute phase performs the call and never fails.

The division into two phases is advantageous for verification. The combined checking phases of all kernel calls are a substantial fraction of the kernel. However, the checking phases do not mutate any kernel state, and thus proving the preservation

of properties across a checking phase is simplified, and benefits from a high degree of proof automation.

The two-phase structure also benefits the execute phase both practically and in verification. Kernel call failures will occur in the checking phase where state is not mutated—the kernel can simply return an error. The execute phase thus avoids both the implementation and proof complexity of having to roll back partially complete operations—it always completes.

3.3. Kernel Memory Management

The seL4 kernel uses a model of memory allocation that exports control of the in-kernel allocation to authorised applications, as introduced in Section 2.1. The model is motivated by the desire to strictly align the in-kernel behaviour performed on behalf of an application to the authority of the application. A consequence is that the kernel heap can be precisely partitioned between applications: each application owns that part of the heap that it holds authority over, including the authority to allocate. This turned out to be essential for expressing and proving the security properties of integrity and confidentiality introduced later in Section 5, which are phrased on top of a formal access-control abstraction that makes this partitioning explicit. Those security proofs thereby formally validate the utility of the kernel's memory allocation model for enforcing isolation between applications.

The core characteristics of the kernel's memory allocation model are as follows.

—Memory allocation is explicit and only performed on retyping an untyped memory object.
—Memory allocation is strictly bounded by the available memory in an untyped memory object.
—Kernel objects are not implicitly shared, or re-used.

While the model enables precise reasoning about memory usage, 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 is correct, not that the user-level policy makes sense. Obviously, moving allocation policy to user-level 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 module may be as simple as statically partitioning the initial memory into isolated subsystems, which can then perform more complex memory management within mutually untrusted subsystems.

The correctness of the in-kernel 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 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, 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.

An additional indirect benefit of explicit memory management is that most kernel calls involve no memory allocation, thus removing one potential execution failure mode. This simplifies the division of each call into checking and execution phases.

3.4. 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. 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—see for instance separate work on a multicore version of seL4 [von Tessin 2010, 2012, 2013]—they are outside the scope of this article. In this article, we focus on uniprocessor support where the degree of interleaving of execution and nondeterminism 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 nondeterministic optional yield. Blocking kernel primitives, such as in lock acquisition and waiting on condition variables, are also a form of nondeterministic 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 [Ford et al. 1999].

Interrupt complexity has two forms: nondeterministic 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, central to our goals for seL4 is a design that lends itself to building real-world safety- and security-critical systems, including systems with hard real-time requirements. Hence, bounded interrupt latencies are important.

Our approach is to run the kernel with interrupts mostly disabled, except for a number of carefully-placed interrupt points. If, in this 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 retries the (modified) operation, including repeating the checking phase and thus re-establishing all the preconditions for continuing 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 [Ford et al. 1999].

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 exceptions are object initialisation, revocation, and destruction, whose operations are either inherently unbounded, or have large finite bounds.

We make these operations preemptable by using an *incremental consistency* design pattern. Correct completion of a preempted operation may be critical to kernel integrity. We therefore cannot store the preempted state in the user-level state (registers) used to restart the system call. Instead, we store the state of progress either in the object itself, or in the last capability referencing the object being destroyed; we term such a capability a *zombie*. This ensures that the kernel moves from one consistent state to the next whether or not a preemption occurs.

A collateral advantage of incremental consistency is the way it deals with concurrent invocations of the same object. If a destroy operation is preempted by another thread invoking the same object, the preemptor will simply continue where the first thread was preempted (a form of priority and time slice inheritance), instead of becoming dependent (blocked) on the completion of the preempted thread. When the originally preempted operation is restarted, it may find its capability invalid and return immediately.

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. Within 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.

3.5. 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.

3.6. 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 by itself 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.

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

Fig. 4. Isabelle/HOL code for scheduler at abstract level.

Nonpreemptable 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 [Heiser 2009] have also adopted an event-based design, motivated by the desire to reduce the kernel's memory footprint (due to the use of a single kernel stack rather than per-thread stacks).

4. FUNCTIONAL CORRECTNESS

This section describes the functional correctness proof of seL4 stretching from the binary ELF file up to the abstract functional specification of the kernel, including a verified IPC fastpath. We explain each of the specification layers as well as the major steps in the functional correctness proof, starting with the abstract specification in Section 4.1. We end this section with a detailed description of the assumptions of this proof.

The functional correctness proof itself has appeared previously [Klein et al. 2009b, 2010], as well as the translation validation step between C semantics and binary executable [Sewell et al. 2013]. The fastpath verification in Section 4.6 is new.

4.1. Abstract Specification

Referring back to Figure 3 in Section 2.3, the abstract specification is the most abstract specification layer that still fully describes the functional behaviour of the kernel: it 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; 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 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, which in seL4 is treated specially. The Isabelle/HOL code for this is shown in Figure 4. 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). As

```
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
```

Fig. 5. Haskell code for `schedule`.

mentioned, we generate the executable Isabelle/HOL specification automatically from a Haskell program. 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 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 5 shows part of the Haskell source of the scheduler specification at this level. The additional complexity becomes apparent in the `chooseThread` function. Without showing the full detail, the important point to note is that it 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 of the kernel correctly implements the abstract specification. With its extreme level of detail, this proof alone already provides strong design assurance. It concentrates the highest level of manual work

and creativity in the overall verification. It also already contains of the order of 90% of the invariants that we have proved about the kernel.

4.3. C Implementation

The most detailed layer in our verification is the C implementation, or more precisely the formal semantics of the C code after it has passed through the C preprocessor. In the initial functional verification of seL4, without verification of the binary output of the compiler, the translation from C into Isabelle was correctness-critical. We therefore took 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 and ARMv7). 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 [Tuch et al. 2007; Tuch 2008, 2009]. 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 impose, 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).

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 assembly 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

```
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;
}
```

Fig. 6. C code for part of the scheduler.

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 [Cock 2008].

Figure 6 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 6 are examples of generated bitfield accessors.

With the translation validation approach to the binary output of the compiler, the C semantics is now not correctness critical any more. Since translation validation is not yet successful for all optimisation levels above GCC -O1, we still are interested in a strongly conservative C parser. However, assuming translation validation, and given a correct ARM binary semantics base model, the binary verification pass will reject any behaviour in the binary that is not exhibited by the semantic representation of the C program. How we arrived at this semantic representation is now irrelevant, we could have written it manually in a way that looks nothing like the original C—as long as the proof that the binary implements it correctly succeeds, Hoare logic properties will transfer by refinement as usual from the higher levels. In fact, even though that has not been the case yet, we could now use the binary verification to soundly reason about compiler bugs. We would merely need to model precisely the same bug on the C semantics level. If the higher-level proofs succeeded this would mean that we would still have achieved the desired kernel behaviour and would have worked around the compiler bug successfully.

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,

```
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
clearExMonitor :: 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
```

Fig. 7. Machine interface functions.

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 usual program semantics. 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 more details, such as the currently masked interrupts. 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.

Figure 7 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 underspecification. 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 underspecified. The less behaviour we prescribe, the fewer assumptions the model makes about the hardware.

In the seL4 implementation, the functions in Figure 7 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 specification layers above. An example is the function `getIFSR`, which on ARM returns the instruction fault status register after a page fault. For this function, whose body is a single assembly instruction, we only assume that it does not change the memory state of the machine, which is easy to check.

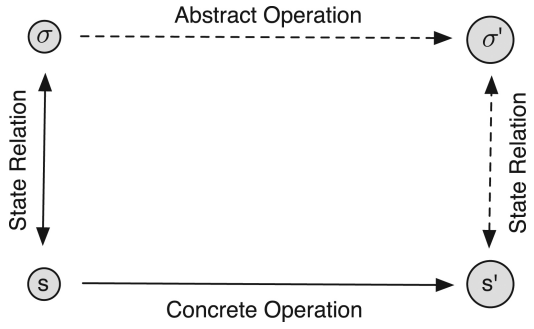


Fig. 8. Forward simulation.

4.5. The Functional Correctness Proof

The functional correctness proof links the C and binary implementations to the executable and abstract specifications. It is the first and most important of the results we have proved about the kernel. The technique we use to express functional correctness formally is refinement. Program *C refines* program *A*, if the behaviours of *C* are a subset of the the behaviours of *A*. A behaviour is an execution trace of a state machine. In seL4, the more precise formal setting is *data refinement* [de Roever and Engelhardt 1998], and these traces consist of user-visible machine state together with additional kernel-internal state. In a refinement step from *A* to *C*, we must preserve the user-visible part of the trace, but are allowed to change the remaining kernel-internal representation of state. This means, the C implementation must only produce traces of interaction with users if the abstract specification can produce the same traces. We have formalised the refinement property for general state machines in Isabelle/HOL, and we instantiate each of the specifications in the previous sections into this state-machine framework so we can relate them to each other uniformly.

We have also proved the well-known reduction of refinement to *forward simulation*, illustrated in Figure 8: 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.

The key proof strategy for seL4 was to split the problem into logically separate subproblems as early as possible. The first such separation was the proof of an *invariant* at every level. An invariant is a property that holds of all encountered states, proved by showing that the invariant is preserved by all possible transitions in the system. The forward simulation step may then assume the initial states satisfy the invariant. The invariants at the abstract and executable layers are detailed descriptions of the allowable states of the kernel, which we will describe further in Section 4.9. Simple invariants on the C and binary levels are used in the binary verification.

Figure 9 shows the proof stack from Figure 3 that makes up the functional correctness proof. The two refinement layers above the C implementation in Figure 9 are proven by hand in Isabelle/HOL. These results compose into a single abstract-to-C refinement proof in Isabelle/HOL. The proof technique for binary verification is different, as we will describe in detail in Section 4.7, but it also produces a forward simulation property,

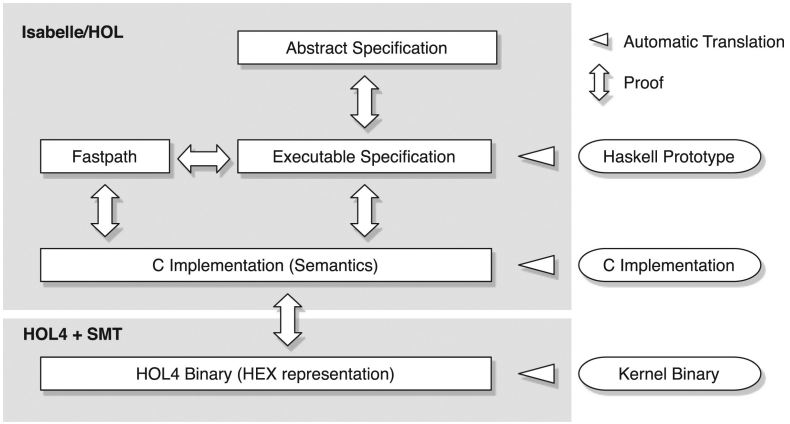


Fig. 9. Functional correctness proofs.

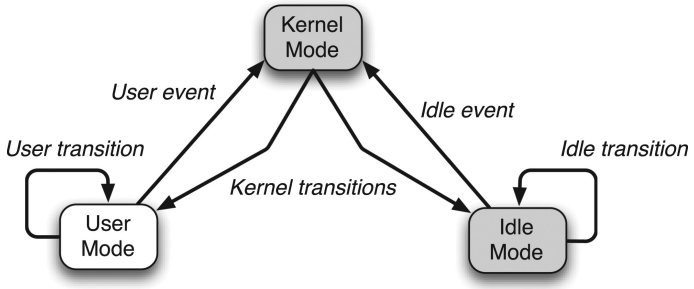


Fig. 10. Types of transitions.

which combined with the previous result forms an end-to-end refinement from abstract specification to kernel binary.

We now describe the instantiation of the general refinement framework to the seL4 kernel. Figure 10 shows the kinds of transitions we have in our state machine: 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. They describe a complete atomic kernel execution, for instance a complete system call. *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 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 Figure 9.

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 nondeterministic 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 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 the following.

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

Clearly the challenging components of these proofs are the ones involving the kernel transitions. We prove these using a further specialisation of forward simulation to apply to program fragments, which we call correspondence. Correspondence establishes simulation and nonfailure of two snippets of program syntax, assuming some preconditions. Correspondence can be used, for instance, to show that the body of a C function performs the same role as its counterpart from the executable specification, assuming the kernel invariants hold initially.

Failure is a notion that exists at each program level with different meanings. Failure in C, for instance, means either an explicit kernel panic, a memory access to an invalid address, or a violation of the C standard, all of which must be ruled out for functional correctness.

The correspondence mechanism is paired with a mechanism for proving invariants over program fragments which assumes nonfailure. Their results compose to give Theorem 1 and Theorem 2. The formal details of these correspondence and invariant proof mechanisms have appeared elsewhere [Cock et al. 2008; Winwood et al. 2009].

Most C functions have an abstract and executable counterpart, and we show two correspondence results (executable to abstract and C to executable). Typically, the bodies of these three functions also have related structure, and contain subcomponents with logically related roles, allowing the correspondence results to be built incrementally out of results for smaller program fragments. The abstract and executable functions will also have one or many invariant proofs. The proofs of these correspondence and invariant results make up the bulk of the functional correctness proof.

The invariant framework allows some flexibility for the proof engineer whether, for each function, to prove all invariants together in one statement or separately in multiple lemmas one at a time. Many abstract functions come with dozens of different invariant lemmas. This is why we chose to keep the nonfailure obligations in the (singular) correspondence proof. In every other respect, we have aimed to move effort from the correspondence proofs to the invariant proofs, which, with code from only one specification level present, are technically more straightforward.

For a typical function, the simplest proof is the correspondence between abstract and executable specification. The C correspondence is more difficult thanks to detailed encodings and syntactic complexities of the C language. In our experience, the invariant proofs are typically the hardest component, requiring significantly more effort than correspondence.

We explain the assumptions of this and the other proofs on seL4 in Section 4.8.

While the effort of the whole seL4 microkernel development, including design, documentation, coding, and testing, was 2.2 person-years (py), the total effort of the seL4 correctness proof was about 20 py: 9 py went into the formal frameworks and tools, and 11 py on the seL4-specific proof (roughly 8 py for the first refinement, including invariants, and 3 py for the second). These effort numbers, their implications and causes are discussed in detail in Section 7.2.

An important aspect of this functional correctness proof is that it focusses on the correctness of the kernel, not necessarily on being convenient for reasoning about

user-level programs. In later security proofs, described in Section 5, it turned out that, in particular for information-flow noninterference, we needed more detail on the constraints of user-level behaviour. Our initial specification assumed any user to nondeterministically change any user-level accessible memory. This is an over-approximation, but it was enough to show safety and functional correctness for the kernel.

For noninterference between users, the top-level refinement state machine did not give enough information about how virtual memory, controlled by the kernel, constrains the behaviour of user applications. As a prerequisite for the noninterference proof, but also for more convenient reasoning about user-level programs on top of the kernel, we strengthened the refinement automaton described previously. We extracted the virtual-memory state of the machine [Daum et al. 2014] and constrained user behaviour by a precise notion of what is mapped when the user is running. We did this for each specification level, thereby also strengthening the assurance about how virtual memory is treated inside the kernel.

4.6. Fastpath Verification

This section describes the extension of the functional correctness proof to include the optional IPC fastpath. The fastpath is a hand-tuned implementation of the IPC (interprocess communication) paths through the seL4 kernel. These IPC paths are used to pass messages between user components, and performance of these paths is considered critical for the usability of microkernels [Liedtke 1993] (performance is discussed in Section 7.1). The fastpath is written in C as an additional C file which is provided as an optional component in the kernel's build system. We show that the functional correctness proof holds whether the fastpath is enabled or not.

This demonstrates that the verification tools we have are capable of handling substantial variations between the implementation and specification. In this case, major differences in the order and structure of code paths are introduced for the sake of optimisation.

The design of the fastpath in seL4 is similar to the one in L4Ka::Pistachio [L4Ka Team 2004] and other L4 microkernels. The fastpath code targets the Call and ReplyWait system calls. We refer to these two code paths collectively as the kernel fastpath. On either system call the kernel entry mechanism calls directly to the fastpath code. The first half of the fastpath checks whether the current situation falls within the optimised case. If so, the second half of the fastpath handles the system call. If not, the fastpath calls back to the standard seL4 system call entry point (sometimes called the slow path), which handles the more general case. This control flow is sketched in Figure 11.

The fastpath targets the case of the Call and ReplyWait system calls where a message will be sent immediately and control will be transferred with the message. For this to occur, another thread must be waiting to receive the message being sent and must be of sufficient priority to continue immediately. The fastpath also requires that the message sent fits in CPU registers and that no capabilities are transferred to the receiver. This is the common case for a client/server pair communicating via synchronous remote procedure calls. The fastpath for the two targeted system calls is implemented in the C functions `fastpath_call` and `fastpath_reply_wait`.

Fastpaths in L4 microkernels are typically implemented in assembly for maximum performance. We implemented the fastpath in C to make use of our existing verification environment. To obtain similar performance, we repeatedly examined the output of the C compiler in search of optimal code. In the process we found that, given sufficient guidance by an expert programmer, GCC (at least on ARM) will produce code that is as fast as the best hand-optimised assembly code [Blackham and Heiser 2012]. This shows that assembly fastpaths are no longer necessary.

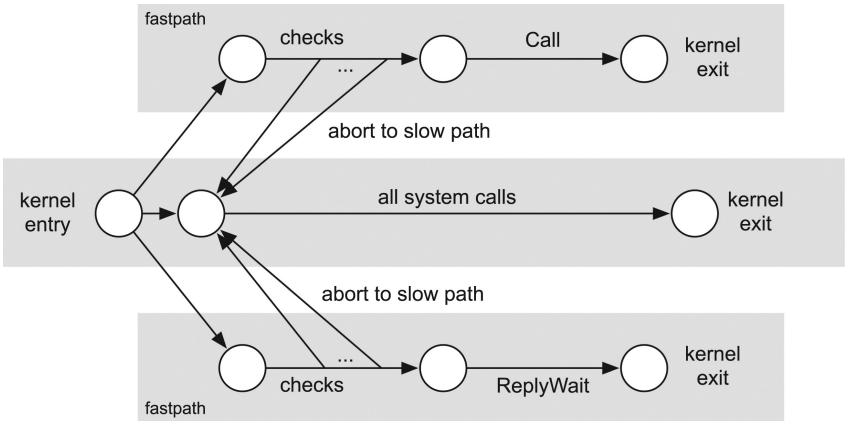


Fig. 11. Fastpath control flow.

The fastpath functions can be extensively optimised for the special case they run in. For example, the capability lookup function used in the fastpath does not need to be able to report errors. If any error is encountered, the fastpath is aborted and the existing kernel implementation repeats the lookup and produces the needed error message. In particular, since the fastpath lookup does not need to distinguish between two different error cases, it can avoid one conditional loop exit.

The proof that the fastpath-extended kernel refines the seL4 specification is divided into the following two subproofs.

(1) We create an executable specification of the fastpath and prove that the C code correctly implements this specification. We developed this executable specification directly in Isabelle/HOL, because there was no need for Haskell-level API prototyping.
(2) We prove that this executable specification of the fastpath refines the previous executable specification of the whole kernel.

Composing these results with the previous refinement (Theorem 1), we prove that the kernel with the fastpath enabled, correctly implements the abstract specification.

The first proof makes use of the existing C refinement framework. The second proof is a refinement between two models at the same abstraction level. This "sideways" refinement makes use of a number of proof techniques we had not previously used. In particular, the fastpath optimisations frequently re-order write operations in the kernel and thereby make it difficult to find execution points where the states of both models are easy to relate to each other.

An example of reordering is the ReplyWait system call, which in the slow path is defined as first executing a full Reply system call followed immediately by a full Wait system call. Both of these system calls will perform capability lookups and argument checks first, and then perform the needed operations. In the slow path, the capability lookup needed in the Wait system call executes in a state in which the Reply step has already been processed. In the fastpath, all the needed checks are collected at the beginning of the combined operation.

To show that this reordering is safe, we prove that none of the capability related operations in the Reply system call can affect the outcome of the capability lookup in the Wait call. This is done by capturing the subset of the capability state on which the lookup depends. We prove that the capability lookup can be viewed as a function of only this subset of the state. Finally, we show that this subset of the state is unchanged

during the Reply procedure, which can be shown as a Hoare triple. These proofs compose to demonstrate that the lookup and check have the same result whether they are executed before or after the body of the Reply procedure.

The fastpath also performs a number of state updates in a different order to the executable specification. This problem can be solved in principle by proving that a number of operations can be swapped with others, under certain side conditions. With enough swap proofs, the fastpath order of operations can be transformed into the standard order. This is conceptually simple but the collection of swaps is tedious to manage.

To simplify this process, we take the executable specifications of the fast path and of the relevant path of the general implementation, and show that each can be decomposed into three programs running roughly in parallel. One of these three programs operates only on the register sets of the sending and receiving threads involved in the system call, another operates only on scheduler state, and the remaining program contains all actions on the rest of the state. Once each specification is split three ways, we then show three pairwise equivalences. This takes care of most of the reordering steps in a uniform manner.

The verification of the main fastpath functions was simplified by a number of functions that are later inlined by the compiler. These function boundaries are not present in the compiler output, but provide helpful decomposition at the logical level. A number of these functions are also shared between the `fastpath_call` and `fastpath_reply_wait`. Even though this reduces the two functions to only 287 source lines of code in total, they still represent the longest contiguous blocks of C code that we verify without decomposition. The total size of the fastpath proof is 5913 lines of proof script, of which about half are spent on refinement to C and half on the verification of the executable model of the fastpath. In terms of effort, the fastpath verification was completed in roughly 5 person months (pm) by one experienced verification engineer. The verification was slightly harder, but of the same order of complexity as the previous proofs.

4.7. Binary Verification

The proof of functional correctness down to the C implementation of seL4 was a significant step. However, this proof still assumed that the C semantics was representative: that the actual behaviour of the kernel would be the C semantics assigned to it. Apart from hardware faults, a compiler or linker defect, a fault in the C-to-Isabelle parser or in the Isabelle C semantics, or a mismatch between the compiler and parser's understanding of the C standard could all lead to a faulty outcome, even with a verified C implementation.

We have investigated two approaches for eliminating these errors. The first is to use the CompCert [Leroy 2006] verified compiler. The second is to compare the semantics of the binary to that created by our C parser directly [Sewell et al. 2013].

As of version 1.10 [Leroy 2012], CompCert translates seL4 with only minor changes to the C code, removing for instance GCC-specific optimisation hints. The exercise of "porting" seL4 to CompCert has made the code base cleaner, less GCC dependent, and more likely to run unchanged with other compilers.

Compiling the kernel with CompCert eliminates the risk of a compiler bug invalidating our correctness result, but it still leaves the possibility of a linker defect, parser flaw or simple semantic mismatch hiding serious problems. We encountered an instance of a linker problem when first using CompCert: We had made assumptions about GCC inside our linker scripts that are not guaranteed by the C standard, and switching to a verified compiler initially made the verified kernel crash! The problem was easy to fix and is of course no fault of CompCert, but it confirms that linker assumptions are critical to correctness. The second, more semantic kind of problem is compounded by the

nature of the C standard, which kernel implementors break on purpose at controlled points in the code. An example is the object rule, which effectively forbids casts of integers to pointers. The seL4 kernel does cast integers to pointers, for instance in page table code, and our C semantics is deliberately permissive in this case. Even though the behaviour of this code is undefined according to the C standard, most compilers will translate it as expected. It is our understanding that the C semantics used by CompCert demand strict conformance to the standard in this regard. This means, even though CompCert translates the code as expected, the assumptions of its correctness theorem would not be satisfied.

The CompCert approach was thus an improvement in the level of assurance over that previously available, but well short of a formal guarantee that composes with our refinement stack (this would be different in application verification that adheres to the standard). To reach a guarantee, some comparison would have to be done between the semantics of the C code in the Coq CompCert and Isabelle/HOL models. Instead of doing this comparison, however, we pursued a more immediate route: comparing the Isabelle/HOL model of seL4's C semantics directly to the semantics of a compiled seL4 binary.

This second approach is a highly trustworthy form of translation validation. To reason about the semantics of a compiled ARM binary program, we use the Cambridge ARM instruction set architecture (ISA) model in the HOL4 theorem prover [Fox 2003; Fox and Myreen 2010]. This ISA model has been extensively validated by comparing its behaviour to that of real hardware [Fox and Myreen 2010]. We used Myreen's approach of decompilation into logic [Myreen 2008] to produce a semantic model of the kernel binary after compilation and linking in the HOL4 theorem prover. The logic of the HOL4 theorem prover is very close to, and compatible with Isabelle/HOL, and automatic translation between these tools is possible.

Using the ARM ISA model in HOL4, Myreen's tool automatically extracts the assembly-level semantics of each function and, by proof, abstracts each compiled C function into its semantic equivalent in higher-order logic. For each C function in the kernel this gives us a higher-order logic representation of its binary-level semantics, as well as our previous higher-order logic representation of its C-level semantics. We can then for each function attempt to prove that the binary-level semantics is a correct refinement of the corresponding C-level semantics. If this is completed for all functions and composed correctly, we gain a refinement theorem between the ELF binary of the kernel and the Isabelle/HOL representation of the C code which can be seamlessly composed with the rest of the refinement stack.

Figure 12 shows the approach in more detail. The proof that the decompiled HOL4 semantics (1) of a function match the C semantics (2) of the same function (the target of the earlier refinement proof) is composed of multiple steps. Starting from the more abstract side, we first transport, by proof, the C semantics (2) to a slightly lower abstraction level (3), where all global variables are part of the addressable reference space. We do this using memory invariants of the C code which we have proved in the previous functional correctness verification. We then convert these functions into an intermediate graph language (4) that models control flow roughly similarly to assembly code. Likewise, we translate the decompiled assembly-level semantics (1), again per function, from the HOL4 prover, via Isabelle/HOL (5), into the same graph language (6). Finally, we prove refinement between the two graph-language representations of each function within an SMT-based logic. For acyclic function graphs, this is broadly straightforward. When loops are present, we search for split points at which the two executions can be synchronised, and prove refinement via induction on the sequence of visits to these split points. A more detailed description of this search is given elsewhere [Sewell et al. 2013].

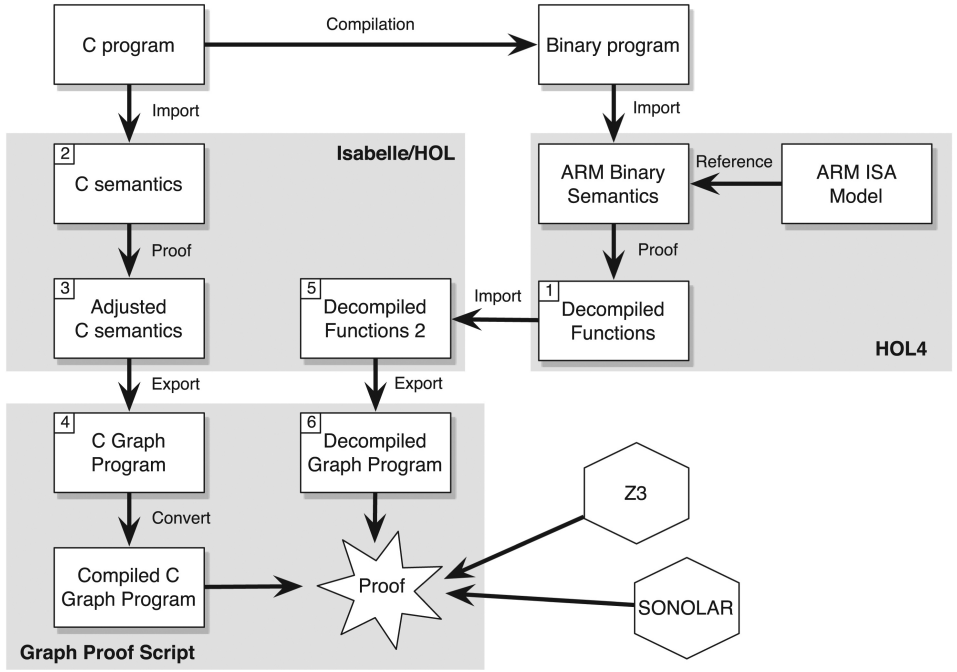


Fig. 12. Binary verification overview for seL4.

We use two SMT solvers, Z3 [de Moura and Bjørner 2008] and Sonolar [Peleska et al. 2011]. We use Z3 first with a small time-out, because it is fast, even though it solves fewer goals. Sonolar is slow but stronger on the theory of arrays and bit vectors which model memory and machine words.

One key aspect of this work is the link between two existing semantics and the link to an existing proof: we join a well-validated hardware model with the Isabelle C semantics and the invariants and assertions that have already been proven on it.

The latter is important, because traditional translation validation approaches must assume what the C standard gives in terms of language guarantees, similarly to the CompCert theorem. Since, as mentioned, OS kernel code purposefully breaks the C standard, these approaches typically fail at such code. Our approach, however, can adapt assertions to fit with existing memory invariants, giving the translation validation proof more information, even where the C standard is followed only partially. Not only does this make the translation validation succeed, it also gives assurance that the compiled code has the expected behaviour, even where it is outside the standard.

Using this approach, we have managed to verify the binary correctness of all seL4 functions that were part of the original verification, when seL4 is compiled with GCC 4.5.1 using the -O1 optimisation level. The verification fails on machine interface functions, which already were assumed correct in the original verification. They make reference to co-processor instructions that are not part of the Cambridge ARM ISA model. The binary verification also fails on the fastpath, which is a recent addition to the verification, and which inlines these co-processor instructions. However, with our per-function approach, we can still assemble the pieces and gain a top-level refinement theorem between C semantics and binary-level semantics of the entire kernel with the explicit assumption that these functions are validated separately.

Table I. Performance of seL4 System Calls with Different GCC Optimisation Levels

	Fastpath Call	Fastpath ReplyWait	Slow path Call	Slow path ReplyWait	ReplyWait with Schedule
GCC -O2	220	234	1522	1453	3017
GCC -O1	304	323	1782	1673	3845
(overhead vs. -O2)	38%	38%	17%	15%	27%

Times quoted in cycles (average of 16 runs after 16 warmups) on the KZM evaluation board, which has a Freescale i.MX31 processor based on a 532-MHz ARM1136 core featuring an 8-stage pipeline.

To support GCC with optimisation level -O2, we have had to make further adjustments to the loop-splitting and refinement process described here, and Myreen has further adjusted his decompilation tool. The main difficulties with increasing optimisation levels are as follows.

—GCC inlines more aggressively, so code from the machine-interface functions now appears in some other function bodies (especially the fastpath) and these functions cannot currently be validated.
—GCC optimises calls beyond the calling convention, so further work has to be done to match assembly-level parameters to C-level parameters.
—GCC performs loop unrolling at -O2, which increases the complexity of the loop-splitting problem.
—GCC optimises stack frame access code, making the stack heuristics employed by Myreen's tool less reliable.

The most recent verification attempt for -O2 succeeds for 214 of 266 function pairs (79%). There are 23 machine interface functions that are impossible for us to cover because they are assigned no C semantics,[2] and code from these functions is then inlined into 15 more function bodies. This leaves us with one nested loop and 13 outright failures (5%).

This figure is a snapshot of a work in progress. The 13 failures we have remaining at the time of writing are essentially deficiencies in the verification tool, and we are confident they can be addressed in the near future through further implementation tuning, with no conceptual changes to the proof process.

We also believe that the performance cost of GCC -O1 over GCC -O2 may be acceptable, at least for very-high-assurance application scenarios. Table I shows the performance cost of using GCC -O1, typically 15–25%. The fastpath is an outlier, possibly because of its large functions or because it has been hand-adjusted for high performance with GCC -O2. The kernel is normally built with GCC -O2, with performance gains at higher optimisation levels considered to be a poor tradeoff for increasing code size.

This result composes with the existing refinement to yield a functional correctness trust chain between high-level security theorems, abstract specification, and binary level. To our knowledge, this is a first. The trust chain rests on the automatic binary verification tool, consisting mainly of the straightforward graph language translations, and the two SMT solvers Z3 and Sonolar, on reading binaries into HOL4, and on the adequacy of the ARM ISA model with respect to real hardware. The ISA model is already extensively validated. Reading binaries into HOL4 is trivial compared to parsing C, since this is not disassembly, but merely reading a string of numbers. Finally, while not achieving the same high standard of an end-to-end LCF-style theorem as in

[2]The fastpath and the initialisation code require some additional machine interface functions that were not previously listed in Figure 7.

the rest of the seL4 verification, the architecture of the binary verification tool already provides strong assurance. Its design is also geared towards enabling full Isabelle LCF-style proofs on the graph language in the future, and hopefully also full replay of the SMT results in Isabelle.

In total, we have replaced our previous assumption, that the compiler and linker execute correctly on seL4 and that the compiler and C-to-Isabelle parser agree on the C semantics, with the new, second-order assumption that the binary verification tool does not exhibit a soundness bug which leads to a missed behaviour mismatch when applied to seL4. This is a huge improvement over trusting the entire compiler and linker implementation.

4.8. Assumptions

With the parser, compiler, and linker assumptions addressed and replaced by much simpler assumptions on binary loading and a well-validated ISA model, the remaining assumptions of the functional correctness proof stay as previously published [Klein et al. 2009b].

We still assume correctness of TLB and cache-flushing operations, as well the correctness of the machine interface functions implemented in handwritten assembly. We also assume hardware correctness. Finally, 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 here and discuss their implications.

The assumptions on the handwritten assembly and hardware interface mean that we do not prove correctness of the register save/restore and the potential context switch on kernel exit. As described in Section 4.4, cache and TLB operations 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 that they are implemented and used correctly.

Hardware correctness is not an assumption that is specific to formal verification. Any programmer must assume that the hardware works as described by the manufacturer, or at least as observed in the past. Nevertheless, it is not a trivial assumption. On the one hand, the errata are often the longer sections in processor manuals. This does not necessarily invalidate a proof, errata just have to be taken into account. On the other hand, hardware can easily be induced to fail by subjecting it to conditions outside its operating parameters. For example, heat development may be a concrete issue in space vehicles or aircraft. The best validated formal theorem will not guarantee correct behaviour if processor and memory are melting underneath.

Formal verification does not exclude traditional measures against such hardware failures, though. For instance, we have observed frequent spurious interrupts on a particular development board the verified kernel operates on. According to the formal model, masked interrupts will not occur, so code that handles such interrupts is dead and should be removed. However, the hardware model can be changed slightly to model this erroneous behaviour of the interrupt controller. It could even be tagged as faulty behaviour so that we can reason about specific parts of the kernel code being executed only in hardware fault modes and prove that the code raises an appropriate alarm.

On ARM processors, in-kernel memory and code access is translated by the TLB. For our C and binary 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 virtual memory 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.

As we have pointed out in previous work, 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, and we have recently made progress on including this part of the kernel in the verification again. For context switching, others [Ni et al. 2007] report verification success, and the Verisoft project [Alkassar et al. 2008] showed how to verify assembly code and hardware interaction. We have also shown that kernel VM access and faults can be modelled foundationally from first principles [Kolanski and Klein 2009; Kolanski 2011].

4.9. Functional Correctness Assurance

Having outlined the limitations of the functional correctness verification, we now discuss the properties that are proved.

Overall, we show that the behaviour of the binary 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 are preserved by refinement on the significantly simpler abstract specification, rather than against the far more complicated implementation. Properties that are preserved by refinement include all those that can be expressed as Hoare triples, as well as some noninterference properties, amongst others. The security properties of Section 5 are good examples—we proved these over the abstract specification, resulting in an estimated effort reduction of at least an order of magnitude.

In addition to the implementation correctness statement, our strengthened proof technique for forward simulation [Cock et al. 2008] implies that $\mathcal{M}_E$, $\mathcal{M}_C$, and the kernel binary 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 in the executable design specification are true on all code paths, and that the kernel never accesses a null pointer or a misaligned pointer. Such assertions can be used to transfer local information from, for instance, the executable specification to the abstract specification, and locally exploit an invariant that is only proved on one of these levels.

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 cannot 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 article 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:

(1) low-level memory invariants,
(2) typing invariants,
(3) data structure invariants, and
(4) algorithmic invariants.

The first two categories could be covered in part 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-node entry containing a reference to a thread control block. The invariant would state that the type of the first object is a capability storage node, that the entry is within array bounds of that node, 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 include value ranges such as using only a certain number of bits for hardware address space identifiers (ASIDs). Often they also 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, such as 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 throughout 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 nonlocal 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 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 subregion 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.

Functional correctness is a very strong property, and tells us that seL4 correctly implements the behaviours specified in the abstract specification. A cynic might say, however, that an implementation proof only shows that the implementation has no more bugs than the specification contains. This is true to an extent: specifications that do not meet users' requirements are a universal hazard for formal verification projects. The best way to reduce the risk of this hazard is to prove further properties about the specification that formally encode user-requirements—that is, to *prove* that the specification, and thus the implementation, meet their stated requirements. We have done this for seL4 for a range of security properties, which we report on in the following section.

5. PROVING SECURITY ENFORCEMENT

One of the primary design goals of seL4 was to provide a foundation for implementing secure systems, by enforcing classic security properties such as authority confinement, integrity and confidentiality. However, it is not enough to simply read the abstract specification in order to convince oneself that seL4 does indeed enforce these properties. Instead, we have proved that it does so. Here, we summarise these proofs, which are reported in full elsewhere [Sewell et al. 2011; Murray et al. 2012, 2013]. They were carried out over seL4's abstract specification and then carried over to its implementation by the formal refinement theorems that embody functional correctness. In performing these proofs, we have made seL4 the first general-purpose kernel with implementation-level

proofs of these classic security properties, realising the 30-year-old dream of provable operating systems security [Feiertag and Neumann 1979; Walker et al. 1980].

Importantly, these proofs tell us not only that seL4 can enforce these properties, but also under which conditions: the proof assumptions formally encode how the general-purpose microkernel should be deployed to enforce a particular security property. We developed an abstract *access-control policy* model for describing access-control configurations of seL4. This describes the authority that each subject has in the system over all others. This model then formed the basis for formally phrasing the security properties of authority confinement, integrity and confidentiality. We begin by describing this model in Section 5.1, before describing the proofs of authority confinement and integrity in Section 5.2 and the proof of confidentiality in Section 5.3. Finally, we discuss the assumptions and limitations of these proofs in Section 5.4.

5.1. Access Control Model

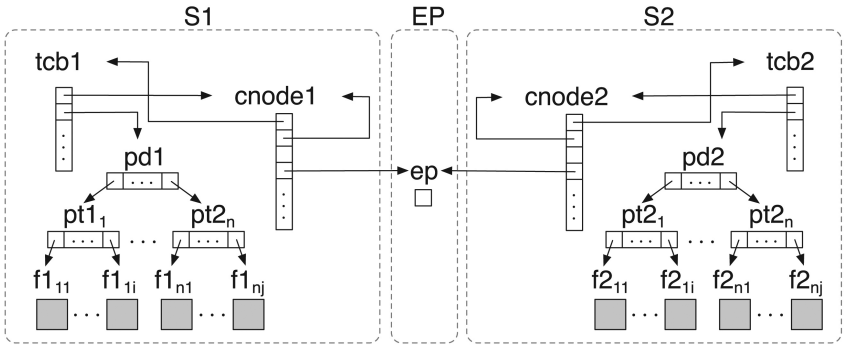
Access control is one of the primary security functions that OS kernels provide. It is used to enforce high-level security properties like authority confinement, integrity and confidentiality on *untrusted* subjects, that is, those that cannot be relied upon to behave securely. To express these properties formally, we construct a model for describing the access-control configuration of seL4. Such a model is necessary because the implementation-level access-control state in a high-performance microkernel such as seL4 is complex; the access-control model provides a usable abstraction for reasoning about such implementation-level configurations. This model is necessarily more nuanced than traditional textbook security models, such as the Bell-LaPadula model [Bell and LaPadula 1976] or traditional take-grant models of capability-based security [Lipton and Snyder 1977], because it reflects the unavoidable complexities and tradeoffs that arise in any high-performance microkernel [Klein et al. 2011].

An access-control system controls the access of *subjects* to *objects* [Lampson 1971], by restricting the operations that subjects may perform on objects in each state of the system. In seL4, the subjects are threads, and the objects are all kernel objects, including memory pages and threads themselves. The part of the system state used to make access control decisions is called the *protection state*. It is this state that our access-control model captures.

In seL4, the implementation-level protection state is mostly represented explicitly in the capabilities held by each subject. This explicit, fine-grained representation of authority is one of the features of capability-based access-control mechanisms. In reality, however, some implicit protection state always remains, for instance encoded in the control state of a thread, or in the presence of virtual memory mappings in the MMU. The protection state governs not only whether a subject is allowed to read or write an object, but also how each subject may modify the protection state. For instance, the authority for capabilities to be transmitted and shared between subjects is itself provided by CNode and endpoint capabilities.

To understand why it is necessary to introduce an access-control model for tractable reasoning about the protection state, consider the example in Figure 13(a) of two threads t_1 (represented by its thread control block, tcb_1) and t_2 (represented by tcb_2) communicating through an endpoint ep, and the question of exactly what authority t_1 has over t_2. For this, we need to consider all possible kernel operations potentially performed on behalf of t_1 and what their effects might be on t_2.

For instance, the low-level (kernel-internal) operation `setThreadState`(t_2, *st*) sets the status of t_2's thread control block to *st*. The obvious case is if t_1 has a thread capability to t_2, that is, if the CSpace associated with the thread control block of t_1 contains a capability pointing to the thread control block of t_2—then t_1 is allowed to change t_2's status. However, there are other cases, such as if t_1 has a send capability to the endpoint ep, t_2



(a) Two communicating threads.

(b) Corresponding access-control policy.

Fig. 13. System objects and corresponding access-control policy.

has a receive capability to the endpoint and t_2 is blocked on this endpoint. Then, if t_1 is performing a send to the endpoint, the internal function `setThreadState`(t_2, `Running`) will be rightly called to change the status of t_2 from `Blocked` to `Running`. What this shows is that the challenge in reasoning about access control in a real microkernel is that the kernel's protection state can be very detailed, and cumbersome to describe formally. It is even more cumbersome to describe precisely what the allowed effects of each operation are at this level.

To address this problem, we make use of the traditional concept of a *policy* to model access-control configurations: a policy can be seen as an abstraction of the protection state. We assign a label to each object and subject, and we specify the authority between labels as a directed graph. We map the concrete access rights in the seL4 protection state into a simple set of abstract authority types such as Read, Write, Grant, described further below. For instance, the policy for our small example in Figure 13(a) of two communicating threads t_1 and t_2 would tag t_1 with label S1, t_2 with label S2 and the endpoint ep they use to communicate could get its own label EP. All objects contained in t_1's CSpace and virtual address space, together with t_1's TCB, are all labelled with S1, and similarly for t_2. The dotted boxes in Figure 13(a), indicate the grouping of objects into labels. The resulting access-control policy is depicted in Figure 13(b).

Capturing the protection state in such a policy simplifies reasoning in three ways. First, the number of labels is typically smaller than the number of objects in the system—depending on the system, a label can contain thousands of objects. Second, when using the policy to reason about untrusted subjects, the policy will typically be static over each system call whereas the protection state will typically change. Finally, we can formulate the allowed effects of operations of untrusted subjects by consulting the policy rather than the more complex protection state.

The access rights between labels in our model are: Receive, SyncSend, AsyncSend, Reset, Grant, Write, Read, and Control. We make a distinction between synchronous and asynchronous send, because the former has different implications for confidentiality as explained later in Section 5.3. The two other nonstandard rights are Reset and Control. The former is the authority to reset an object to its initial state for re-using resources. The latter confers the authority to completely determine the behaviour of

the object, for example, by writing to a thread's registers. The Create right known from the classical take-grant models [Lipton and Snyder 1977] is subsumed here by Control.

Access Policy Refinement. To give meaning to this model, we need to relate it to the implementation of the kernel, which in our case means relating it to the abstract specification of seL4. In particular, we need to define formally when an access-control policy, such as in Figure 13(b), is consistent with a state of the seL4 abstract specification such as in Figure 13(a)—that is, when the authority in the system state is consistent with that specified by the policy. Concretely, we say that a state s refines a policy p when the authority of each subject in the state does not exceed its authority in p. This relation is determined by the policy graph, the abstraction function from state to policy, and the current subject. We capture it formally in the predicate $\mathsf{pas\_refined}(p, s)$, where "pas" stands for *policy, abstraction, subject*. This predicate effectively decodes the complex protection state in s, and checks that the authority present in s is also present in p. $\mathsf{pas\_refined}(p, s)$ additionally places *wellformedness* constraints on the policy p. We describe these constraints in Section 5.2. They restrict the policy to sensible access-control configurations in which it makes sense to reason about high-level security properties like authority confinement, integrity and confidentiality.

5.2. Authority Confinement and Integrity

5.2.1. Authority Confinement. Authority confinement is a general security property on capability-based systems where authority may change dynamically. The property gives a bound to this dynamic behaviour: It says that, given a policy, one can statically determine the maximum authority each subject can acquire. Conversely, given a policy p that claims to be a maximum authority bound, authority confinement says that no subject can gain more authority than described in p. In take-grant systems, for instance, this maximum bound is simply the reflexive, transitive closure of all take and grant capabilities [Lipton and Snyder 1977].

For seL4, we can formally express authority confinement directly on top of the predicate $\mathsf{pas\_refined}$ defined in the previous section. We want to assert that, if all authority in the current state s is captured in the policy p, then, for all kernel calls leading to any state s', the authority contained in s' will still be captured in the same p. More formally, if $\mathsf{pas\_refined}(p, s)$ holds, then $\mathsf{pas\_refined}(p, s')$ will also hold for all directly reachable, future states s'. That means, we need to prove that $\mathsf{pas\_refined}$ is an invariant of kernel execution.

However, authority confinement stated as above will clearly not hold for all policies p: for instance, the policy could explicitly allow one subject to give authority to another. This means, authority in that other subject could grow. We therefore need to capture which policies make sense for authority confinement. Formally, $\mathsf{pas\_refined}(p, s)$ asserts that p must be wellformed, which means p must meet the following constraints:

(1) the label of the current subject, which is stored in p, must not have Control authority to any separate label,
(2) there must be no Grant authority between separate labels, and
(3) every label must have full authority to itself.

The second constraint, as alluded to above, is necessary to enforce authority confinement, and corresponds to transitivity in the classical take-grant model. The third corresponds to reflexivity in take-grant. The first simply ensures that the partitioning of subjects and objects between labels is meaningful. Control authority from subject s_1 to subject s_2 would mean that s_1 can make s_2 do anything on s_1's behalf.

Note that the first condition only need apply to the subject that is currently executing, according to p, not to all subjects in the system. This is a difference to traditional

take-grant models. It allows us to apply the authority confinement theorem selectively to only the untrusted subjects in the system where access control must be mandatory. Trusted subjects may have more authority than allowed for authority confinement, but can be verified separately not to abuse this authority. That means, while all subjects in the system may become the current subject at some point, we may only want to apply the access-control theorem to some of them.

The formal theorem of authority confinement, proved over the seL4 abstract specification, has two further side conditions.

THEOREM 4 (AUTHORITY CONFINEMENT AT ABSTRACT LEVEL). *Given an access-control policy p, a kernel state s, and a kernel state s' reached after performing one transition step of $\mathcal{M}_A$, then if* pas_refined(p, s) *holds initially,* pas_refined(p, s') *will hold in the final state, assuming the general system invariants* invs *and* ct_active *(current thread is active) for noninterrupt events in s, and assuming the current thread in s is the current subject in p.*

The two predicates invs and ct_active are re-used from the functional correctness proof between $\mathcal{M}_A$ and $\mathcal{M}_E$. We have shown in this previous proof that invs is invariant over kernel execution, and that ct_active holds for any noninterrupt event at kernel entry as required, so the theorem integrates seamlessly with the previous result.

Note that the only side condition that the user of this theorem needs to check manually is wellformedness of the policy p, which is phrased purely in terms of the policy abstraction. That means, no knowledge of the implementation-level protection state is required. All other conditions are either already proved invariant or can be established by user-level system initialisation as described in Section 6.1.

5.2.2. Integrity. Integrity is the security property that says a subject cannot modify the system state without explicit authorisation to do so.

We again use the high-level access-control policy p to capture the authority in the system and to describe which state modifications are authorised by this policy. The resulting theorem can then be used to reason about kernel calls on behalf of untrusted subjects. The theorem will tell us precisely which parts of the system state are guaranteed to remain unchanged, and which parts of the system state can change. As with authority confinement, this information is already captured in the abstract specification of seL4. The integrity theorem merely makes it significantly easier to understand and easier to apply when reasoning about systems on top of the kernel.

The formalisation again builds on the predicate pas_refined(p, s) from Section 5.1, and we phrase the allowed modifications in s in terms of the policy p. The grouping of subjects and objects into access-control labels allows us to make broad brush strokes at the level of the policy, instead of referring to the lower-level kernel state. For instance, when applying integrity reasoning, one is interested in the changes a subject may make in all other subjects, not in the changes the subject may make to itself. Phrasing integrity in terms of the policy easily expresses that by saying a subject may change anything within its own label.

In formalising integrity, we are interested in expressing the difference between the states s and s' before and after any kernel call, according to a policy p. We write integrity(p, s, s') to capture this relation, and we can use the predicate pas_refined(p, s) as before to assert that the state s conforms to the policy p. In more detail, the integrity predicate integrity(p, s, s') then says for each authority in p from the current subject over any other subject, what state modifications it authorises. For instance, a Send capability may lead to a change in the thread state of any subject that owns a Receive capability on the subject with the corresponding endpoint. A Write capability allows state modifications to the page of memory it authorises, but no more. A Read capability

to memory allows no state change at all, and so forth. In essence, this gives us an over-approximation of kernel behaviour. When we apply the theorem in analysing a concrete system, we will conclude that not even this over-approximation includes state changes to, for instance, the area of memory we are interested in protecting, and so no actual kernel execution will either.

The formal integrity theorem, proved for the abstract specification, again has side conditions that are already satisfied from the functional correctness proof.

THEOREM 5 (INTEGRITY AT ABSTRACT LEVEL). *Given a policy p, a kernel state s and a kernel state s' reached after a transition of $\mathcal{M}_A$, the property* integrity(p, s, s') *holds, assuming that* pas_refined(p, s) *holds, that the general system invariants* invs *and* ct_active *for noninterrupt events hold in s, and that the current current thread in s is the current subject in p.*

5.2.3. Code-Level Theorems. The proof of integrity and authority confinement was completed in the space of 4 months with a total effort of 7.4 pm. The proof comprises 10,500 lines of Isabelle script. The effort is dramatically reduced compared to the functional correctness proof, because the proof could be completed on the abstract specification instead of the code-level of seL4. However, composed with the functional correctness result, the properties automatically hold for the code as well! More precisely, using Theorem 3, we have the following two end-to-end theorems at the source code level.

THEOREM 6 (AUTHORITY CONFINEMENT AT CODE LEVEL). *For any policy p, the property* pas_refined *is invariant over all transitions of $\mathcal{M}_C$ between observable states, under the same assumptions as in Theorem 4.*

THEOREM 7 (INTEGRITY AT CODE LEVEL). *The property* integrity(p, s, s') *holds over all transitions of $\mathcal{M}_C$ between observable states s and s', under the same assumptions as in Theorem 5.*

The theorems further transfer down to the binary level in the same way, albeit not as end-to-end theorems in a single proof system.

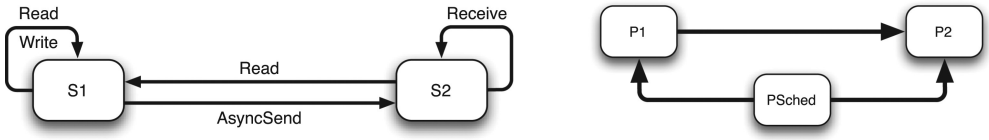
These two properties in conjunction allow us to establish bounds on the execution behaviour of untrusted user-level components by inspecting the policy only. For instance, for a suitably restrictive policy p, using the predicate integrity(p, s, s'), we can directly conclude that all memory private to other components of the system is unchanged between s and s', no matter what code the untrusted component is running or what kernel call it tried to perform. Using authority confinement, we can reason that the policy remains consistent with the protection state and so the same argument holds for the next system call as well. The two properties thus compose over whole system executions to ensure that untrusted components cannot affect state outside their authority.

The massive reduction in effort is the true power of the abstraction provided by the functional correctness proofs. Any additional property that is preserved by refinement can be established more rapidly and with less effort by proving it over the abstract specification. This massive saving of effort was repeated during the following proofs of confidentiality for seL4.

5.3. Confidentiality: Information-Flow Security

Confidentiality is the dual of integrity, and informally states that no subject may read information to which it does not have read authority. In this section we summarise the proof that seL4 enforces confidentiality (reported in full elsewhere [Murray et al. 2012, 2013]). It builds on the previous integrity and authority confinement proofs.

Phrasing confidentiality turns out to be more involved than integrity and authority confinement, for two reasons. Firstly, unlike those earlier properties, confidentiality



(a) Example access-control policy. (b) Corresponding information-flow policy.

Fig. 14. Access control policy and corresponding information-flow policy.

applies to entire execution traces of the system not just to single transitions. In this sense, it is a global *whole-system property* that applies during the entire lifetime of a system after initialisation, and so requires more machinery to state formally.

The second reason is that the effect of reading some piece of state is not directly visible: to infer that component A read some state associated with component B, during some transition from a state s to a resulting state s', we cannot simply look at the difference between s and s'. The difference can tell us only that A's state somehow changed—it cannot tell us what information A has learned when that state-change occurred. To work that out, we must consider a second hypothetical execution beginning from some initial state in which B's internal state differs from that in s and see whether this causes a difference in the resulting state of A. This is the essence of reasoning about confidentiality and is captured formally by the classical security property of *noninterference* [Goguen and Meseguer 1982]. To establish confidentiality, we proved a variant of noninterference for seL4 that we introduce shortly. Noninterference is a security property that compares pairs of executions of a system, unlike the security properties seen so far which talk only about a single execution at a time.

Formalising Confidentiality. Confidentiality is often expressed formally by asserting that all flows of information in a system must be in accordance with the system's *information-flow policy*. A static information-flow policy divides the system state between a fixed set of security partitions, and defines the allowed information flows between those partitions. Figure 14(b) depicts an example information-flow policy that contains three partitions, P1, P2 and PSched, and asserts that information may flow from PSched to all others, and from P1 to P2, but that no other information flows are permitted between these partitions.

These information-flow policies are on yet another abstraction level from the access-control policies discussed so far—they only talk about information flows, not access rights. However, we can easily compute an information-flow policy from an access-control policy. As in integrity, our noninterference property works for any policy that conforms to the system state, that is, while the policy is static for each system, the kernel can be configured to enforce any wellformed policy. As an example, Figure 14(a) shows the access-control policy from which the information-flow policy in Figure 14(b) was derived.

Generally, to construct the information-flow policy for a system, we associate each subject S_i in the system's access-control policy with a distinct partition P_i in the information-flow policy. The computed information-flow policies for seL4 additionally always include a distinguished partition PSched, which captures the kernel state associated with the scheduler. Formally, the information-flow policy is a binary relation $\rightsquigarrow$ on partitions: we write $\mathsf{P}_i \rightsquigarrow \mathsf{P}_j$ if and only if information is allowed to flow from partition P_i to partition P_j.

For each system, we derive the information-flow policy $\rightsquigarrow$ mechanically from the system's access-control policy, following three simple rules. Briefly, the access-control policy defines what we call the *extent* of each non-PSched-partition. This is simply the

part of the system state that the policy allows the partition to observe. This includes the state the partition can read directly as well as any state the kernel reads during a system call and then reveals to the partition. The rules for calculating $\rightsquigarrow$ then say the following.

(1) $\mathsf{P}_i \rightsquigarrow \mathsf{P}_j$ if the access-control policy allows subject S_i to affect any subject in P_j's extent.
(2) PSched $\rightsquigarrow \mathsf{P}_i$ for all P_i, and PSched $\rightsquigarrow$ PSched.
(3) No other information flows are permitted.

These rules ensure that $\rightsquigarrow$ never allows any partition, other than PSched itself, to send information to PSched. This is important since PSched is allowed to send information to all other partitions. Indeed, it does so whenever the scheduler schedules a thread in any such partition. Thus, the rules for mechanically deriving $\rightsquigarrow$ always prevent the scheduler from becoming a global transitive channel for information flow.

To assert that all information flows permitted by the kernel are present in the information-flow policy $\rightsquigarrow$, we use a variant [Murray et al. 2012] of *intransitive noninterference* [Haigh and Young 1987; Rushby 1992]. Like integrity and authority confinement, but unlike many traditional noninterference definitions, this noninterference variant is preserved by refinement. This means we can prove it about the kernel's abstract specification $\mathcal{M}_A$, and then conclude that it must hold for $\mathcal{M}_C$, that is, for the code, by Theorem 3.

Informally, our noninterference property aims to show that the information that flows to each partition comes only from that part of the system that $\rightsquigarrow$ says may affect it. For a partition P, this part is its own extent, plus the extent of whichever partition Q is currently running, if $Q \rightsquigarrow P$. To decide this, we take two states of the system that are equal with respect to P and Q, but that are potentially wildly different in any other part of the system. We then run for a single step of execution from these two different states and analyse the two resulting states: they should be indistinguishable to P, that is, be equal for all state in P's extent, although they may be different anywhere else. If this is true for any such states, we can conclude the resulting information flow to P does not violate $\rightsquigarrow$.

We lift this notion to multiple execution steps, and denote the resulting property nonleakage, following von Oheimb's terminology [von Oheimb 2004], by which our definition is inspired. While we omit its formal definition here, and refer the interested reader elsewhere [Murray et al. 2013], we note that the resulting formulation supports *intransitive* noninterference policies. A classic example of such a policy has three user partitions, PHigh, PDowngrader and PLow; and allows PHigh $\rightsquigarrow$ PDowngrader and PDowngrader $\rightsquigarrow$ PLow but does not allow the direct flow PHigh $\rightsquigarrow$ PLow to ensure that all information flowing from high to low must pass through the downgrader. In contrast to classical noninterference formulations, our nonleakage formulation also permits the current partition to depend on the dynamic state of the system–which is necessary to apply it to a kernel like seL4, in which the currently running partition is defined by the internal scheduler-state [Murray et al. 2012].

We proved nonleakage for $\mathcal{M}_A$ after some modifications to the kernel. In particular, proving confidentiality requires us to show that the scheduler's decision about which partition to execute next never depends on the state of any other partition besides PSched (recall that $\rightsquigarrow$ forbids information flows from any other partition to PSched). This behaviour was not enforced by seL4's original scheduler, so we modified it to implement *partition scheduling*: it uses a static round-robin schedule for choosing *between* partitions, while allowing priority-based scheduling of threads within partitions. This

involved updating the kernel's code and its various specifications together with the functional correctness proof to capture this new scheduling behaviour.

We also updated the kernel specifications to carefully separate the actions of the scheduler PSched from those of the other partitions. This required no code changes. In the automaton of Figure 10, scheduling actions occur within the actions that handle partition system calls, as part of the transitions from kernel-mode back to user- or idle-mode. Separating out scheduling actions into their own transitions of the automaton resulted in a more complicated construction, the details of which are elsewhere [Murray et al. 2013]. Making these changes for the scheduler gave us new artefacts $\mathcal{M}_A$, $\mathcal{M}_E$ and $\mathcal{M}_C$ for which we repaired Theorems 1 and 2, and so Theorem 3.

In order to be preserved by refinement [Murray et al. 2012], nonleakage tolerates no *partition-visible* nondeterminism in $\mathcal{M}_A$. This is because such nondeterminism here allows all secrets to be revealed. In the example of Figure 14, consider the the trivial specification S that nondeterministically sets the value of P1's variable `v1` to either `0` or `1`. Suppose S is implemented by the following insecure program P that includes P2's variable `v2` and leaks its value in `v1`:

```
if (v2) v1 = 0; else v1 = 1;
```

P is a valid refinement of S, and so any confidentiality property that is preserved by refinement must judge S to be insecure, because it has insecure refinements.

To prevent such cases, we removed the user-visible nondeterminism in $\mathcal{M}_A$. We avoided duplicating the original abstract specification by making it *extensible* [Matichuk and Murray 2012]: nondeterministic operations in the original specification are replaced by placeholders. By defining specification code to fill these placeholders, one creates an *instantiation* of the extensible specification. We defined two instantiations: one corresponding to the original nondeterministic specification $\mathcal{M}_A$ and the other $\mathcal{M}_A^D$, which replaced the nondeterministic operations with abstract versions of their deterministic implementations from $\mathcal{M}_E$, removing all of the partition-visible nondeterminism in $\mathcal{M}_A$. This allowed us to have both the original nondeterministic specification and the deterministic one, without unneeded duplication between the two.

It was important to keep the nondeterministic specification, because it allows us to experiment with different implementation choices in the future (like alternative secure schedulers), plus it enables simpler reasoning about properties that do not require determinism. We repaired Theorem 1 for $\mathcal{M}_A^D$ and proved confidentiality of $\mathcal{M}_A^D$, allowing us to conclude confidentiality of $\mathcal{M}_C$. From this result, it follows that nonleakage holds of $\mathcal{M}_C$. Again, the result carries down to the binary level, albeit not as an end-to-end theorem in a single proof system.

THEOREM 8. *Let s_0 denote the initial state of $\mathcal{M}_C$ and p be an access-control policy that captures the protection state in s_0. Let $\rightsquigarrow$ be the corresponding information-flow policy and* $\mathsf{nonleakage}_C$ *denote* nonleakage *with respect to $\rightsquigarrow$ applied to $\mathcal{M}_C$. If the kernel invariants hold for s_0 and the policy p is consistent with the authority in s_0 and is suitably well formed, then* $\mathsf{nonleakage}_C$ *holds. Formally,*

$$\mathsf{invs}(s_0) \wedge \mathsf{pas\_refined}(p, s_0) \wedge \mathsf{pas\_wellformed\_nonleakage}(p, s_0) \longrightarrow \mathsf{nonleakage}_C.$$

Here, $\mathsf{pas\_wellformed\_nonleakage}(p, s_0)$ asserts some extra well-formedness conditions on the access-control policy p and the initial state s_0. In particular, we require that all nontimer interrupts are disabled in s_0 and that no authority exists in p to re-enable them. This is because seL4 does not properly isolate the interrupts of one partition from the others, so its primitive interrupt-delivery facility must be restricted from use for confidentiality to hold. In consequence, device drivers must poll for pending interrupts via memory-mapped IO. We also require that no partition has the authority to destroy

any partition-crossing resources, which could be detectable by other partitions that share those resources and therefore provide a back-channel for communication in violation of $\rightsquigarrow$. The pas_wellformed_nonleakage(p, s_0) predicate encodes this assertion as a restriction on the access-control policy p. Neither of these restrictions are uncommon in high-assurance separation kernels [Murray et al. 2013].

As with the earlier proofs of integrity and authority confinement, we gained a substantial saving in effort by proving confidentiality over the abstract specification and then transferring this result to the C code by refinement. The proofs of confidentiality were completed over a 21 month period, and required about roughly 40.7 pm of effort. This includes implementing the partition scheduler ($\approx$ 1.8 pm), making the abstract specification deterministic, and repairing the functional correctness proofs ($\approx$ 18.5 pm), as well as the proofs of confidentiality themselves ($\approx$ 20.4 pm). While proving confidentiality required about 5 times the effort of proving integrity and authority confinement, much of this stemmed from having to make the abstract specification deterministic. We estimate that trying to prove it directly of the C implementation would have required at least the 20 py of the original functional correctness proof. So proving it about the abstract specification, despite having to make it deterministic, was the right thing to do.

5.4. Security Assumptions and Assurance

The corpses of security proofs, broken by novel exploits, litter the history of computer security. Proofs break when they are not logically correct, their assumptions are unrealistic, or the property proved does not mean what one thought it did. We assess the strengths of our security proofs for seL4 against each of these criteria, and, in doing so, highlight their assumptions and the level of assurance they provide.

As explained earlier, we can have great confidence that our proofs are logically correct, since they are carried out in the LCF-style proof assistant Isabelle/HOL, and all derivations are from first principles. Proof correctness is, therefore, a nonissue in practice for these proofs.

Our security proofs build on top of the functional correctness proofs, and so inherit their assumptions which were detailed earlier in Section 4.8. The integrity and confidentiality proofs additionally assume that the system has been correctly configured in accordance with the access-control policy p. We discuss how to ensure this in Section 6.1. The confidentiality proof makes a few further assumptions about the system configuration mentioned earlier: that all nontimer interrupts are disabled and that no partition has the authority to destroy any partition-crossing resources. These force device drivers to poll for interrupts and prevent partition-crossing communication channels from being destroyed, respectively.

The security proofs also make a number of extra-logical assumptions, besides those inherited from the functional correctness proofs. We effectively assume that direct memory access (DMA) is disabled, which is a common restriction for separation kernels. For confidentiality, our formulation assumes that the static round-robin partition schedule is allowed to be globally known, so we do not prevent one partition from knowing about the existence of another, for instance. This implies that all partitions can learn about the passage of global time: each time a partition is scheduled it can infer exactly how many timer ticks must have elapsed because this is precisely specified by the static schedule. Our confidentiality property also assumes that user-space partitions have effective access to only those user-space sources of information that are present in $\mathcal{M}_A^D$: machine registers and memory pages mapped with read rights. We need this because we model user-space partitions as deterministic functions of these inputs. This implies the assumption that any other sources of information exposed by the platform must be correctly cleared by the kernel on each partition switch.

Each of these assumptions are reasonable for high-assurance systems, and many are amenable to validation or even formal analysis; however, this is currently left as future work.

The integrity theorem is extremely strong: if it says that the contents of some memory cannot be changed by some thread, then we can be sure that the memory will remain unchanged under the proof's assumptions. However, making a similar claim for confidentiality with respect to inferring memory contents is less straightforward, because here we must consider *covert channels* below the level of abstraction of the formal models of the kernel that may allow information to be inferred despite the confidentiality theorem.

This theorem, while being the strongest evidence produced so far that a general-purpose OS kernel can enforce confidentiality, is not an iron-clad statement of security. In particular, there are known covert channels in seL4 not covered by this theorem. For instance, because none of our formal models talk about time, nonleakage says very little about the absence of timing channels. Also, our formal machine model says little about low-level platform state, such as caches etc., and so nonleakage cannot reason about storage channels arising from this state. Finally, there could be remaining states, that is, storage channels, below the level of abstraction of $\mathcal{M}_A^D$. However, and perhaps surprisingly, these create channels only if the kernel never reads from such state. This is because nonleakage *does* force us to reason about any potential channel below the level of abstraction of $\mathcal{M}_A^D$ if that channel is ever read by the kernel and then its value exposed to some partition: such a channel would show up as partition-visible nondeterminism. However, if there are channels below the level of $\mathcal{M}_A^D$ that the kernel never touches, then nonleakage does not force us to prove anything about such channels. An example might be an undocumented feature of the hardware platform that is unused by the kernel. If such a channel exists, and the kernel fails to clear it on a partition switch, then this could provide a means for partitions to communicate covertly despite Theorem 8.

Such covert channels must be addressed by complementary techniques. The strength in our security results is that they first enable such channels to be more easily identified—we can use the formal statement of the security theorems and their assumptions to guide any security review. Second, they allow whole classes of attack to be ruled out, such as information leakage to confined subjects without access to timing sources for Theorem 8.

Ultimately, we see these security results as a step in a larger vision that will for the first time enable security proofs of whole systems built on top of a verified OS kernel. seL4 is now the world's first—and only—general-purpose kernel that provably enforces the classic high-level security properties of integrity and confidentiality. This is further evidence of its suitability for the construction of high-assurance systems.

6. BUILDING TRUSTWORTHY SYSTEMS ON SEL4

Up to this point, the focus of this article has been the seL4 kernel itself together with the verification of its functional correctness and security properties.

In this section, we turn towards building trustworthy systems on top of seL4, and the additional properties and analyses that are required from the kernel to do so. In particular, we present a method for provably bringing a system into a known configuration and protection state in Section 6.1, and we describe a sound worst-case execution time profile of seL4 in Section 6.2, which is required in real-time critical systems.

6.1. Capability Distribution Language and System Initialisation

The security guarantees presented in the previous section apply to a running system, ensuring that a security property is preserved during execution. This assumes that the

property is initially established, that is, it assumes the presence of an initial system state that corresponds to a given high-level access-control and information-flow policy. More generally, our approach to building trustworthy systems is to minimise the trusted computing base by a componentised architecture, where untrusted components can be isolated from trusted ones by a careful distribution of authority. Again, the critical step here is initialising the system into a state satisfying the specified architecture. In this section, we describe the language we developed to describe such a desired initial state [Kuz et al. 2010], and the automatic, verified initialiser program we use to set up a system in a given configuration [Boyton et al. 2013].

The access-control policy model of seL4 introduced in the previous section is at an abstraction level that provides just enough detail to conveniently reason about authority confinement, integrity and confidentiality. However, it does not contain enough detail to describe how a seL4-based system should be configured at runtime to enforce a particular access-control policy. This requires knowing, for instance, not just the abstract authority that one subject has over another, but the specific concrete capabilities that confer this authority. For instance, the access-control policy may allow Control authority over a certain thread, but it does not specify if that Control authority is achieved by a direct capability to the thread's TCB, or by access to its capability storage. Likewise, it is not enough to know that a subject has Read authority to a shared memory page, we also need to know at what virtual address the page should appear in the subject's virtual address space, and so on. The *capDL* [Kuz et al. 2010] level, which constitutes the remaining verification layer of Figure 3, bridges this gap.

The *capability distribution language* capDL is a language for describing kernel configurations solely in terms of objects and capabilities. Next to this language with its formal syntax and semantics, we have constructed an additional specification of seL4 that describes the behaviour of the kernel on capDL states. We call this the *capDL kernel model* or capDL kernel specification, denoted $\mathcal{M}_D$. It is more abstract than the abstract functional specification, but less abstract than the access-control policy model. Descriptions of the configuration of particular systems, written in capDL, are called *capDL system descriptions*, which if the context is clear are called just *capDL descriptions* for short.

The capDL language describes system states abstractly, prescribing which objects are present in the system and which capabilities they possess, but not which further internal state these objects have. As mentioned in Section 5, and as in any real system, some authority in seL4 is represented not explicitly in capabilities but instead implicitly in the system state, for instance in virtual-memory mappings and authority-relevant thread state. CapDL describes all protection state in terms of capabilities, making implicit protection state explicit and easier to reason about. The capDL language therefore contains more capabilities and capability types than seL4 itself. For instance, it treats such virtual-memory mappings and authority-relevant thread state as special capabilities. In other words, capDL describes the complete protection state of the system, at a level of detail sufficient to initialise the system into a particular configuration.

Given a capDL system description, we synthesise the input data structures for a user-level program for initialising the system into this configuration, together with a formal proof of correctness of this initialiser's design. The initialiser is the first user-level task to run after boot time, with full authority to all available memory.

Figure 15 shows the process of using a capDL description to produce a correctly initialised system: Given a complete capDL description of a system, enriched with information on what code should be attached to which threads, we automatically initialise a system into a state that conforms to the input description. The user-level initialiser iterates through the capDL description of the desired initial state, creates

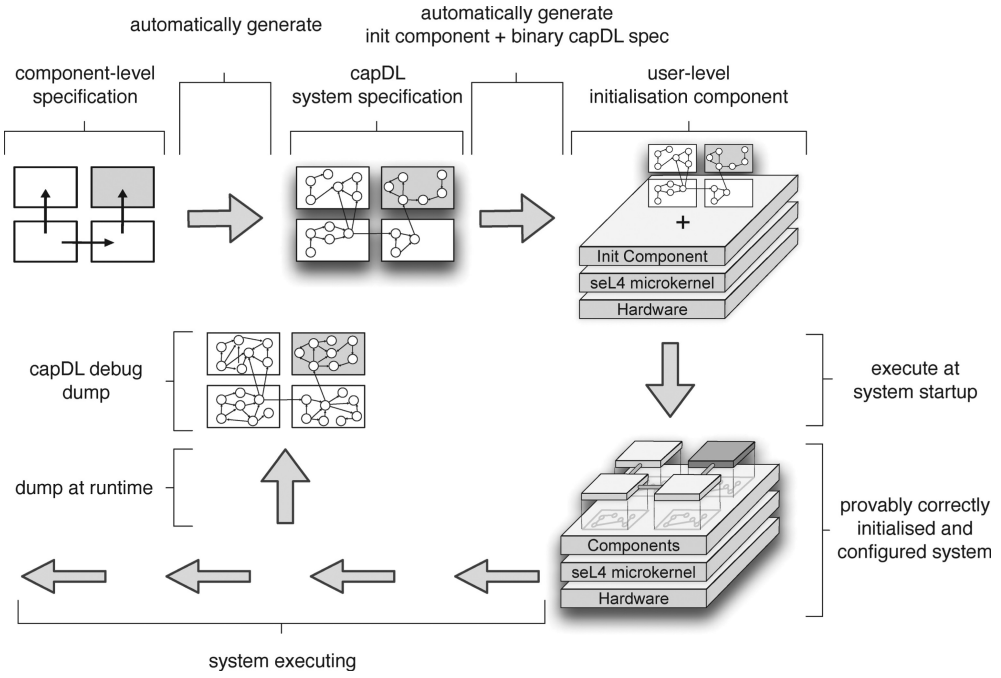


Fig. 15. Using capDL for correct system initialisation.

the appropriate objects, gives them the desired authority, connects them up with the appropriate binaries, and sets them up to run. When this component finishes execution at system startup, the system will be left in a state that corresponds to this desired capDL state and can commence normal execution of user-level components.

We have constructed a functional design model for this initialiser, and have proved its main correctness property: it will initialise the system into a state whose capDL abstraction is isomorphic to the desired capDL description [Boyton et al. 2013]. An additional step in future work would be to prove that the C implementation of the initialiser implements its functional design model correctly. The initialiser correctness theorem is the following.

THEOREM 9. *Let $\mathcal{I}(D)$ be the initialiser model applied to a capDL system description D. If D is well formed, and the system starts in a state s_b directly after boot time, then the execution of $\mathcal{I}(D)$ started in s_b either aborts and halts the system, or, if successful, the state s_0 after the execution of $\mathcal{I}(D)$ isomorphically conforms to D such that the same objects and capabilities are present in both D and s_0 modulo renaming of physical addresses.*

Well formedness of D excludes system configurations that are not achievable, in particular configurations that cannot occur in any execution of the kernel, for instance because they violate kernel invariants. The property also encodes a number of current practical limitations in the initialiser, such as supporting delegation of untyped capabilities [Boyton et al. 2013].

To prove this initialisation correctness property, it is convenient to specify the initialiser $\mathcal{I}$ in terms of kernel behaviour on the level of the capDL kernel model $\mathcal{M}_D$. To do so, we proved Theorem 10, which states that the capDL model of seL4 is correctly implemented by the kernel.

THEOREM 10. *The abstract specification $\mathcal{M}_A$ of seL4 refines the specification of kernel behaviour on the capDL level $\mathcal{M}_D$.*

This theorem says that the capDL kernel model, which specifies kernel behaviour in terms of capDL states, is correct. Composed with the previous Theorem 3, we get that the C code is a correct refinement of $\mathcal{M}_D$, and with binary verification also the binary. Further composed with Theorem 9, this theorem ensures that the initialiser correctness proof interacts with the kernel behaviour as implemented by the real kernel.

The proof proceeds as usual in refinement: we define a correspondence relation between capDL states and the abstract functional specification of seL4 and then prove that it is preserved in forward simulation. The difference to our previous refinement proofs is that the capDL model concentrates on only the protection state of the system and contains a significantly larger degree of nondeterminism. It does not contain information about the contents of memory, for instance. With this amount of abstraction, the capDL level alone would not be sufficient to prove functional correctness of user-level components. For configuring a system, however, it provides precisely what is necessary.

To initialise a system in accordance with a particular security policy, for instance an information-flow policy, one first describes how the system should be initialised using a capDL description. To check that this configuration will enforce the policy, we apply a sound mapping from capDL to the access-control policy model, and from there use the results of Section 5 to check that the obtained access-control configuration is consistent with the security policy to be enforced. The initial capDL description of the system need not be written by hand, but can be generated by a high-level component-based toolchain [Kuz et al. 2010], which can automatically check that the generated capDL description corresponds with the policy to be enforced. Theorem 11 states that the mapping from capDL description to the access-control policy is sound and indeed covers all relevant protection state in the system.

THEOREM 11. *Translating a kernel state from the abstract level to the access-control policy level is equivalent to translating it first to capDL and then to the policy level.*

This theorem enables security reasoning starting from capDL system descriptions used for system initialisation, as just explained. It allows one to map a capDL description, which describes a state of the system, to the corresponding access-control policy. This mapping produces the same result as if we had mapped directly from the kernel state to the access-control policy. It implies that a capDL state description captures all information relevant to the protection state of an access-control policy, that is, instead of having to know the precise memory content of the machine, it is enough to reason about the information present in a capDL description to apply the security theorems described in Section 5.

Besides allowing provably correct system initialisation, the capDL language also facilitates run-time system debugging, as indicated in Figure 15. Specifically, we use capDL to describe protection state dumps of running seL4 systems if they are compiled for debug mode. This enables system developers to inspect the capability state for debugging and error analysis. Such systems dumps, as well as generated or hand-written capDL descriptions, can be visualised as graphs that have kernel objects as nodes and capabilities as edges. As illustrated in the example in Figure 16, visualisation makes it easy to see clusters and, for instance, determine if a particular component allocates a large amount of specific objects. The graph is remarkably dissimilar to the function call graph of seL4 in Figure 1. It clearly shows clusters of different, separate components that are connected through small interfaces. This is the idea: concentrating complexity once in the microkernel so that modular programming becomes possible at

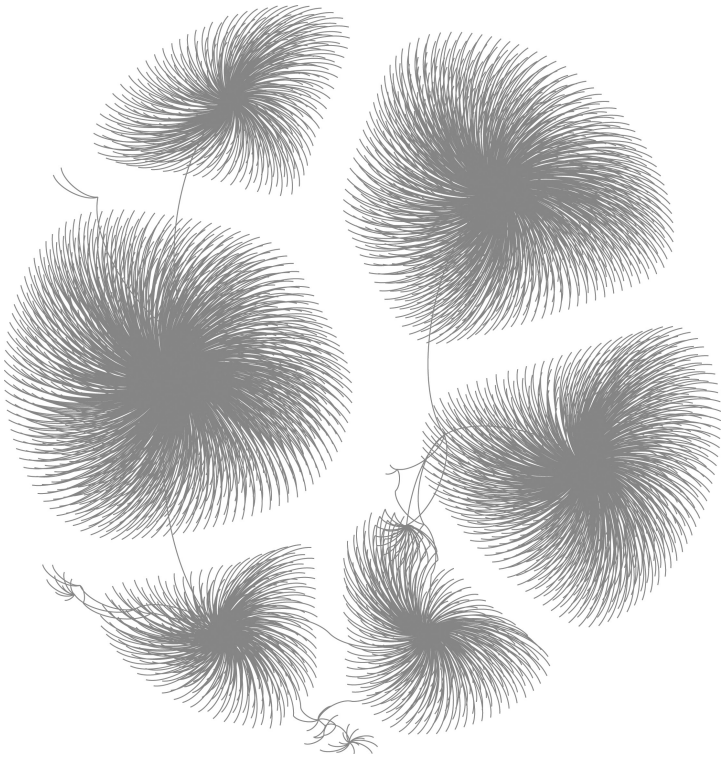


Fig. 16. Visualisation of a capDL dump from an example system.

user level. The image in Figure 16: is simplified for print. In our tool, different object types can appear colour coded, for instance, and the graph is interactively zoomable and expandable.

Figure 16 depicts a snapshot of a running seL4-based system, called the *secure access controller* (SAC) [Andronick et al. 2010]. The snapshot was taken just after system initialisation. The system at this point in time contained three components; each "island" in the figure forms some sub-part of a component. The small bottommost island that connects the two larger ones directly above it is a timer component; it sends notifications to the two connected components, which are a router manager (left) and the SAC controller (right). These larger components each possess a number of Untyped memory capabilities, so that they can dynamically allocate new memory as they run, which is why they are much larger than the timer component. The remaining four topmost islands are parts of the address spaces of the router manager and the SAC controller. These address spaces are large, because each contains an entire GNU/Linux instance that the component in question starts up later in its execution. The SAC controller shares an Endpoint with the router manager, allowing the former to send messages to the latter, which is why those two islands are connected in the figure. Those two components do not, however, share any memory between their address spaces, as the figure shows.

In summary, the theorems in this section allow us to correctly initialise seL4-based systems into a known state that enforces a particular desired security policy. In addition, the capDL language and its visualisation tools are useful development tools in practice.

6.2. Timeliness Analysis

6.2.1. Timeliness Requirements. Functional correctness and the noninterference properties discussed previously are requirements for building provably secure systems. For many safety-critical systems this is not enough. Many such systems are of a real-time nature and need guarantees about the timeliness of operations.

Examples are medical implants and avionics. While in the past such systems were typically implemented on dedicated microcontrollers, using a simple real-time executive without memory protection or dual-mode execution, mushrooming functionality is creating a demand for *mixed criticality* systems [Barhorst et al. 2009], where highly critical (often life-critical) functionality is located on the same processor as less critical code.

Highly critical code is subject to strict assurance and certification requirements. It is usually also small and relatively simple, which increases the likelihood of correct operation. Less critical code, however, tends to be more complex and is subject to less stringent or no certification. In a mixed-criticality system it must be possible to certify the most critical code without considering any of the less-critical code.

Mixed-criticality systems are therefore only possible if the highly critical code is strongly isolated from any less-critical code. The isolation is as critical as the highly-critical code itself, and therefore subject to the same assurance and certification requirements.

seL4, as described so far, provides strong *spatial* isolation. To make it an acceptable platform for mixed-criticality systems, it must also provide strong *temporal* isolation. Specifically, it must be possible to analyse the timeliness of execution of the highly critical code without making any assumptions on the behaviour of the less critical code.

This requires an analysis of the temporal behaviour of seL4. We can assume that the highly critical code executes at a higher priority than any less-critical code, so it cannot be preempted. However, an interrupt which invokes highly critical code must be delivered to its user-level handler with a bounded latency, no matter what code is executing at the time the interrupt is raised. We therefore need to establish an upper bound on the worst-case interrupt latency (WCIL).

Interrupts can happen during user-mode or kernel-mode execution. The worst-case latency of an interrupt happening in user mode is the worst-case latency of entering interrupt mode, plus the worst-case execution time of the kernel's interrupt handler, including the delivery of the interrupt message to the user-level handler. However, if the interrupt happens during kernel execution (where interrupts are disabled except for a small number of carefully placed interrupt points, see Section 3.4), this time is extended by the worst-case latency of reaching the next preemption point or regular kernel exit.

6.2.2. Approach. WCET analysis is well established in the real-time community [Wilhelm et al. 2008], and there exist a number of research and commercial analysis tools. However, WCET analysis is normally performed on critical application code, or the short critical sections of a fully-preemptible real-time executive. WCET analysis even of single-mode real-time kernels has proven difficult [Lv et al. 2009b], and, prior to seL4, the only public evidence of WCET analysis of operating systems is for single-mode real-time executives without support for virtual memory. Industry practice is still to measure interrupt latencies under high load and apply a safety factor.

Furthermore, any previously analysed systems inevitably are fully preemptible, meaning that the code paths to be analysed are very short, and very pessimistic assumptions can be made, such as simply adding up the worst-case latencies of all instructions in the critical section.

For a nonpreemptible kernel, such as seL4, some of the code paths that must be analysed are quite long, thousands of instructions. Simply adding up all worst-case

instruction latencies would lead to massive pessimism of orders of magnitude. Such pessimism is not tolerable for many battery-powered devices, especially medical implants, which must operate autonomously for years.

We found no existing tools which would scale to the complexity of code and length of the nonpreemptible code paths in seL4. It is possible that some of the commercial tools might scale, but they tend to only support low-end microcontrollers that are typically used in deeply embedded systems (running unprotected real-time OSes).

We therefore had to build our own WCET tool chain [NICTA 2013b]. We adapted the open-source Chronos framework [Li et al. 2007], added a control-flow graph generator for ARM binaries, and implemented a model of the ARM architecture and the ARM1136 [ARM Ltd. 2005] and ARM A8 pipelines [Blackham et al. 2011]. Chronos is based on the implicit path enumeration technique [Li et al. 1995], which extracts from the control-flow graph an integer linear programming (ILP) problem. We use an off-the-shelf ILP solver (ILOG CPLEX from IBM) to solve these for the WCET. This approach is explained in detail elsewhere [Blackham et al. 2012a]. Chronos also performs a cache analysis, which determines whether a memory access is a guaranteed hit, a guaranteed miss, or undecided (and handled pessimistically).

The analysis virtually inlines all function calls in order to extract a complete execution path. This tends to create many infeasible paths, which lead to an order-of-magnitude pessimism if not eliminated. Furthermore, the analysis needs worst-case iteration counts for all loops, many of these cannot be determined by local static reasoning. In line with current practice, we manually determine loop bounds and infeasible paths and feed this information into the analysis. While, we have made significant progress in automating this (error-prone) process, or at least proving its results correct, this is not yet complete [Blackham and Heiser 2013].

A number of earlier kernel-design decisions were instrumental to make this analysis feasible. The event-based model (see Section 3.4) helps as it hides the complexity of context switching, while seL4's memory model (Section 3.3) avoids much of the complexity that normally comes with virtual-memory management.

6.2.3. Kernel Changes for Improving WCET. Although seL4 had been designed to be real-time capable (see Section 3.4), the original implementation failed to follow the appropriate design pattern in a number of places, and the initial analysis produced worst-case latencies of around one second!

Some of this was easy to fix, by ensuring that the incremental-consistency pattern was used throughout, and placing preemption points in appropriate places; deletion code was usually the culprit. The obvious value of the WCET tool is that it provides guidance to this process, and in many cases, we ended up with a better implementation, although in other places, the need to place preemption points made the implementation more complex (and requires significant reverification effort).

An example of the latter is a specific case of partial deletion: the revocation of a badged capability (see Section 2.1). When a badge is revoked, any pending IPC operations using that badge must be aborted; this means removing from the relevant endpoint's message queue any IPCs using that badge. As the revocation must be preemptible, other threads may in the meantime keep using the endpoint (with different badges). The situation is further complicated by seL4's principle of not performing dynamic memory allocation: all of the revocation state that needs to be retained across preemptions must be stored in the zombie capability.

Another source of long latencies was a classic L4 implementation trick called *lazy scheduling* [Liedtke 1993]: a thread that blocks during an IPC is not removed from the scheduler's ready queue, as it is likely to become unblocked soon (e.g., when a server replies to a request). When the scheduler is invoked at the end of a time slice, it removes

Table II. Computed and Observed Worst-Case Execution Times for the Various Kernel Entry Points (from Blackham et al. [2012a])

Entry point	Computed	Observed	Ratio
System call	436 μs	81 μs	5.4
Undefined instruction	77 μs	43 μs	1.8
Page fault	78 μs	41 μs	1.9
Interrupt	45 μs	14 μs	3.1

all blocked threads from the ready queue, until it finds one that is actually runnable. In the worst case, this operation is limited by the number of threads in the system.

Here the solution was a redesign of scheduling, dubbed *Benno scheduling* [Blackham et al. 2012a], which has the same average-case behaviour while avoiding the pathological worst case of lazy scheduling: When a thread unblocks during IPC, it is not immediately entered into the ready queue (as it may soon block again). Like lazy scheduling, this avoids queue manipulations during IPC, but now the cleanup is simple and O(1): when the time slice expires, only the preempted thread may have to be moved (from an endpoint waiting queue to the ready queue).

The implementation is broken up into small changes and affects almost every scheduler interaction in the kernel. In particular, as mentioned previously, it critically interacts with IPC code. In addition to breaking a small number of existing invariants, the changes led to a new global kernel invariant, which states that all threads in the ready queue are runnable. Proving correctness of the new implementation and establishing the new invariant required a significant amount of work, 9 pm.

Some changes introduced to improve WCET are also beneficial for best- or average-case performance, and should arguably have been used from the beginning, but were originally shunned to keep the implementation simple. An example is the scheduler queue, where we introduced the standard technique of a priority bitmap to speed up the search for the highest-priority thread.

At the time of writing, not all of these changes have been verified. The scheduler bitmap is presently unverified, but is expected to be straightforward as the data structures and algorithms are simple and the changes well localised. The badge revocation changes are also yet to be verified, and while we have no doubt that this is tractable, it will require a significant effort.

6.2.4. Results. Table II shows the resulting WCETs of the kernel. We obtained these on the same KZM evaluation board that was used for the results of Table I.

The rows in the table represent the longest nonpreemptible path segment in the code reachable from each of the kernel's entry points: the standard system call trap, the undefined-instruction trap, the page-fault, and the interrupt handler.

In the table, the "computed" value refers to the output of our WCET analysis (with infeasible paths eliminated). It represents a *safe upper bound* on the true WCET, while the "observed values", obtained from measurement on the board, represents a *safe lower bound*. The difference between the two values has two possible sources: the "observed" value might be too optimistic (because we failed to trigger a higher-latency path) or our analysis too pessimistic. It is most likely the latter (analysis pessimism), although, for safety, we have to assume the former.

Sources of pessimism are the cache analysis and pipeline modelling. Our processor has a split 4-way L1 cache with "random" replacement. We must model this pessimistically, assuming a direct-mapped cache of 1/4 capacity. We can force the actual cache into this configuration (by locking three of the four ways with useless content), which has negligible effect on the observed execution time. Hence, modelling of the cache

replacement policy is not a significant source of pessimism. However, modelling of cache content in Chronos is imperfect and a possible source of pessimism.

Other forms of caching potentially also add to pessimism. We disabled the branch prediction unit, since it is underspecified on our hardware, and actually worsens WCET, as the latency of a mis-predicted branch is higher than the latency of a branch with prediction disabled. We locked the kernel window in the TLB to avoid TLB misses in the kernel.

Modelling timings of the processor core is inevitably pessimistic, as it is underspecified in the documentation. Many instructions are documented as having a range of execution times, with no insight on the microarchitectural state this depends on, we therefore always have to assume the worst case.

For use in mixed-criticality systems, the most interesting timing figure is the worst-case interrupt latency (WCIL). This is the sum of the latency of the longest non-preemptible section (in the system-call handler) and the latency of the interrupt handler itself. At present, the safe estimate for this is 481 μs, while the actually observed latency is 95 μs, a factor-five pessimism.

Interrupt latencies of the order of half a millisecond are tolerable for many but not all hard real-time systems. Commercial real-time OSes typically claim WCIL of the order of a few microseconds, but these kernels do not provide virtual memory, usually not even any form of memory protection. Also, given the industry-standard approach to WCET analysis (i.e., based mostly on testing), there is typically significant doubt about the soundness of such claims.

Using our tool chain, we performed a WCET analysis of a commercial RTOS supporting virtual memory, QNX [QNX 2012]. We compared this with what we think is achievable in seL4 if the requirement for verification was dropped. We found that it should be possible to get the WCIL of the nonpreemptible seL4 kernel to within 50% of that of the fully preemptible QNX (60 μs vs 41 μs) [Blackham et al. 2012b].

We conclude that, in principle, the design of seL4 lends itself to real-time performance that is comparable to that of commercial real-time OSes which have been designed with real-time latencies as the primary concern. However, in order to achieve this, we would have to introduce preemption points in many of the more complex operations in the kernel, resulting in extra invariants, most of which would be hard and costly to prove.

We therefore accept for now that our interrupt latencies are about an order of magnitude larger than what seems achievable. However, our experience also shows that one should be highly sceptical about the safety of any real-time operating system (of otherwise comparable capabilities) that claims significantly lower WCIL: such a system's implementation is unlikely to be much simpler than that of seL4, and if we find our preemption invariants hard to prove, we expect such a system hard to get free of concurrency bugs (especially if it is fully preemptible).

7. EXPERIENCE AND LESSONS LEARNT

7.1. Performance

IPC performance is the most critical metric for evaluation in a microkernel in which all interaction occurs using variations of IPC, including interrupt delivery. We have evaluated the performance of seL4 by comparing IPC performance with L4, which has a long history of data points to draw upon [Liedtke et al. 1997]. The IPC value commonly reported is the best-case number of cycles consumed in kernel mode to deliver a zero length message. Cache misses are usually avoided by careful fastpath construction. This represents the upper bound (lowest cycle count) of IPC performance. Cache contention and increasing message size will result in higher cycle counts. Unlike previously reported L4 kernels, seL4 has an asymmetric IPC path with a reply IPC

being slightly more complex than a send. We report the average of the two directions when comparing seL4 with previously reported one-way results. Note the best-case reported here was obtained with multiple runs using performance counters to confirm the absence of cache conflicts and TLB misses.

Publicly available performance for the Intel XScale PXA 255 (ARMv5) is 151 in-kernel cycles for a one-way IPC [L4HQ 2007]. Our experiments with the open-source OKL4 2.1 [Open Kernel Labs 2008] on the platform we are using (Freescale i.MX31 evaluation board based on a 532MHz ARM1136JF-S which is an ARMv6 ISA) produced 206 cycles as a point for comparison. Both of these kernels use a hand-crafted assembly fastpath.

We previously reported 224 cycles [Klein et al. 2009b] for a one-way IPC for the (at the time unverified) optimised C fastpath. This figure fluctuated over time. It initially increased to 259 cycles, when the formal verification of the IPC fastpath uncovered a small number of corner-case defects. Fixing these required adding additional argument checking, leading to an increased IPC cost.

Further optimisation has reduced the cycle count of the verified fastpath to 227 cycles. Our best unverified performance to date is 188 cycles. There is no fundamental reason why the optimisations required to get there cannot be verified, but that does not necessarily mean they will be. The gap in performance between verified and unverified is pragmatic—the IPC fastpath is generally a moving target that is dependent on the specific processor generation, and also includes several minor optimisations of varying invasiveness for usually small gains. Thus, the performance of the verified IPC fastpath lags as optimisations mature or become broadly applicable, or as a specific processor is targeted. More generally, there is a diminishing return on re-verification costs.

With the benefit of four years of experience, the conclusion remains the same as previously reported: verified seL4 performance is close to that of the fastest L4 kernels.

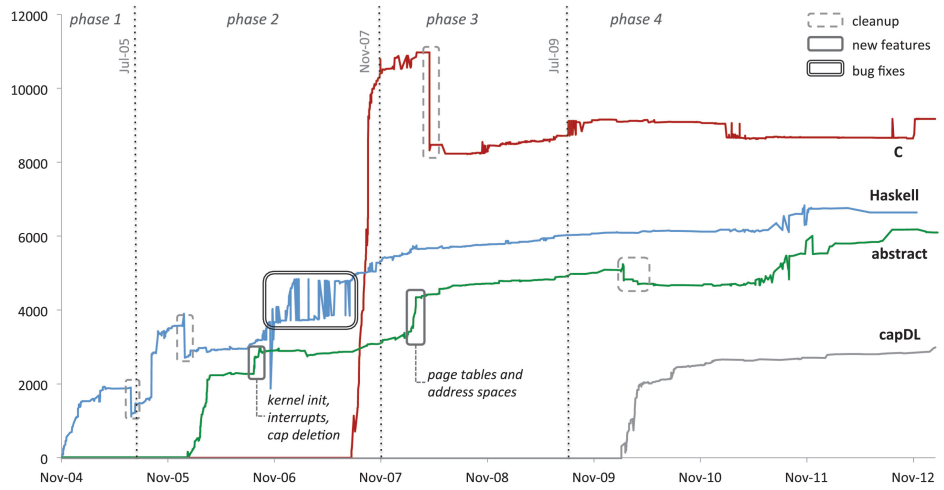
7.2. Verification Effort

This section discusses lessons learnt from analysing the effort of the kernel development and its associated formal proofs. First, we look at the effort of developing the seL4 kernel itself, and in particular the impact of the constraints of verifiability. Secondly, we analyse the main influencing factors in the effort of its proofs of functional correctness, optimisation and security. Finally, we analyse the balance in benefits gained in terms of bugs found and level of assurance achieved.

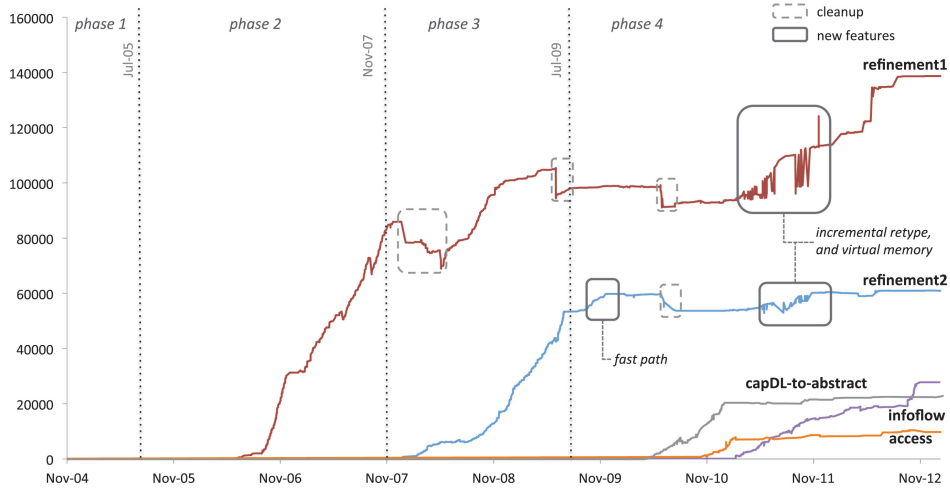
7.2.1. Process and Project Phases. Before looking into effort distribution, and in order to understand how the effort numbers associated with different parts of the project may relate to each other, we first show how the different phases of the project can be observed in version control commit statistics.

The kernel development and its proof of functional correctness were conducted in parallel, as overlapping and mutually dependent activities. This was enabled by the seL4 design process, using a Haskell prototype from which the implementation can be derived and that can be formally linked to the abstract specification, as described in Section 2.2. We conducted this work (kernel development and correctness proof) in three phases, each involving both the kernel and the verification teams.

The phases are observable in the evolution of the sizes of code, models and proofs, represented in the graphs of Figure 17. We obtained these graphs by counting, for each artefact, and for every commit in the version control system from November 2004 up to December 2012, the total number of actual lines of code/model/proof, excluding any empty lines, comments, Isabelle documentation, etc. The time frame reaches into an initial pilot project for seL4 at the beginning, covers the actual design and functional correctness proof from April 2005 to July 2009 and stretches over the development



(a) Size of code/spec artefacts (X: time; Y: SLOC)



(b) Size of proofs (X: time; Y: SLOC)

Fig. 17. Size of code, specs and proofs.

of new security proofs as well as the binary translation validation. We used SLOCCount [Wheeler 2001] for C and Haskell code, and a similar, home-brewed counting tool for Isabelle models and proofs that ignores comments and white space.

In earlier work [Andronick et al. 2012], we had analysed the project from a process point of view, where similar graphs were provided, but for the functional correctness proof only, using a raw lines-of-code count (LOC). The sizes of the four representations of the seL4 kernel (C source code, Haskell prototype, Isabelle abstract specification, and the capDL-level specification of the kernel in Isabelle) are represented in Figure 17(a). The sizes of the two refinements of the correctness proof (*refinement1* relating the Haskell code to the abstract specification, and *refinement2* relating the C code to Haskell) are represented in Figure 17(b) (together with the capDL refinement proof, and the security proofs, enlarged in Figure 18).

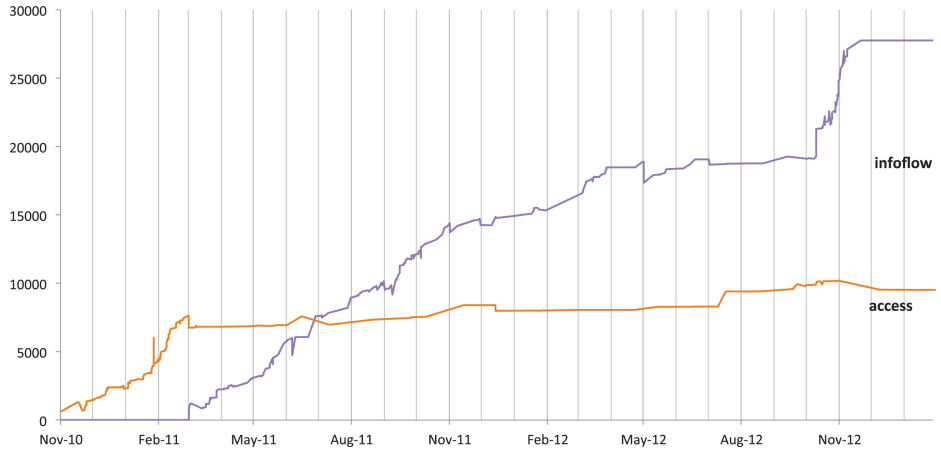


Fig. 18. Size of security proofs—detailed.

In a first phase, the kernel team designed and implemented an initial kernel with limited functionality (no interrupts, single address space and generic linear page table) in Haskell. This is represented by the steep increase in SLOC in the "Haskell" graph in Figure 17(a), followed by a cleanup reducing the line count. During this time, the verification team mostly worked on the verification framework and generic proof libraries (not shown in the graphs). In a second phase, the verification team developed the abstract specification (first steep increase in the "abstract" graph in Figure 17(a)) and performed the first refinement, between this abstract specification and the initial kernel design (large increase in *refinement1* graph in Figure 17(b), up to end of phase 2). During this time, the development team completed the design and Haskell prototype (second increase in the "Haskell" graph, again followed by a cleanup phase), and then later wrote the kernel C implementation (very steep increase in the *"C"* graph in Figure 17(a)).

In addition, as the first refinement was revealing design bugs (i.e., mismatches between the design in the Haskell prototype and the abstract specification), those bugs were fixed in the Haskell prototype. Such fixes are sometimes completed in a separate branch in the version control system, then merged back to the main repository when satisfactory (explaining the oscillations in the "Haskell" graph during bug fixes). Each larger new feature that was added to the kernel Haskell prototype in order to complete the design was gradually introduced in the abstract specification. Note that this process could occur in parallel to other activities and in particular is spread over multiple phases. For instance, we clearly see two main feature additions in the "abstract" graph in Figure 17(a); in phase 2, this corresponds to the addition of interrupts, capability deletion, and kernel initialisation; in phase 3, this corresponds to the addition of page tables and associated API calls.

The third phase consisted of extending the first refinement step to the full kernel (increase in "refinement1" graph in phase 3) and performing the second refinement (steep increase in "refinement2" graph). The big drop in the *"C"* graph in phase 3 is an optimisation in our bitfield generation tool [Cock 2008], mentioned in Section 4.3, to now only generate functions used in the kernel.

The correctness proof for the full kernel was completed in July 2009. This event characterises the transition to the fourth phase of development. The two refinements that make up the functional correctness proof entered a maintenance phase where we keep the proof up-to-date with kernel development. Later, this was complemented by

further work (capDL, security properties and binary verification) which added further verification artefacts but also led to further specification and implementation changes. Each modification to the kernel or to its formal models requires re-establishing all of the formal proofs that are impacted by the change.

Note that we have integrated all proofs into an automated proof checking suite, similar to an automated regression-test suite, but using machine-checked formal proofs instead of executable tests. This provides an automatic check, after each commit into the version control system, of the state of all the existing formal proofs, and identifies which specific portions of the proof must be re-established.

We created the capDL specification in two steps, first defining an initial, comprehensive subset, and then adding the more complex operation of capability deletion (second increase in the "capDL" graph in Figure 17(a)). The proof of refinement with the abstract specification happened largely in parallel, incrementally verifying the operations as they were being defined in the capDL specification.

The access-control model, the proof of authority confinement, and the proof of integrity, were all developed together, indicated by the *access* graph in Figure 18. Work on these proceeded at about the same time as the initial capDL refinement proof. The spike in activity around February 2011 reflects a "sprint to the finish" in order to meet a paper submission deadline.

A similar effect is present in the confidentiality proofs, captured by the "infoflow" graph in Figure 18, at November 2012. The bulk of initial work here was performed by November 2011, at which point the proofs were complete except for those parts of the kernel with user-visible nondeterminism (see Section 5.3). We spent the next 10 months or so making the kernel's abstract specification deterministic.

Once we had removed the user-visible nondeterminism from the abstract specification, we extended the confidentiality proof to cover more of the kernel API, until it eventually covered the entire kernel around October 2012. We completed this proof in November 2012, spending the last month or so on tying together the confidentiality lemmas about each of the internal kernel APIs to produce the top-level confidentiality theorem for the whole system. This involved constructing a more elaborate system automaton (explained in detail elsewhere [Murray et al. 2013]) and, as well as phrasing the confidentiality statement, proving a number of invariants over it. The dramatic spike in output during October 2012 indicates this.

During phase four, the kernel team also worked on further development of the kernel API and various optimisations, mainly the fastpath optimisation and changes targeted at improving the kernel's worst-case execution time. In this phase, kernel modifications were generally tested in an experimental branch of the kernel source. Committing them to the main-line kernel implies an obligation for reverification, which is only done after discussions with the verification team.

For instance, as shown in Figure 17, the fastpath induced reverification mainly in the second refinement (first increase in "refinement2" graph in phase 3). The low impact of such implementation optimisations on the design verification (first refinement) shows that the process achieves a separation of concerns between the design and the implementation.

Large changes (highlighted as "new features" in both "refinement1" and "refinement2" graphs) were due to larger API changes (e.g., a model of virtual memory, or changing the retype operation to be incremental), as well as the production of the extensible and the deterministic specifications required by the information-flow verification.

Figure 19 summarises the effort required for the development of various proofs about the seL4 kernel. Effort is given in person-years (py) required to complete each task; for smaller tasks, we provide person-months (pm). We have recently undertaken rigorous effort counting for some of the activities, analysing every contributor's effort

	Total Effort	Artefacts	Effort
Development Effort	2.2 py	Haskell	2 py
		C implementation	0.2 py (2 pm)

(a) Overall Effort for seL4 Development

	Total Effort	Artefacts		Effort
Correctness Proof Effort	20.5 py	Generic framework & tools		9 py
		seL4 formal models	Abstract Spec	0.3 py (4 pm)
			Exec. Spec	0.2 py (3 pm)
		seL4 formal proofs	Refinement 1	8 py
			Refinement 2	3 py

(b) Correctness Proof Effort

	Total Effort	Artefacts	Effort
Optimisation Proof Effort	0.4 py *	Fast Path	0.4 py (5 pm) *

(c) Optimisation Proof Effort

	Total Effort	Artefacts		Effort
Security Proof Effort	4.1 py *	Integrity		0.6 py (7.4 pm) *
		Confid.	Scheduler Update	0.2 py (1.8 pm) *
			Determinising Spec and Updating Proofs	1.5 py (18.5 pm) *
			Confidentiality Proofs	1.7 py (20.4 pm) *

(d) Security Proof Effort

	Total Effort	Artefacts	Effort
Binary Verif. Effort	2 py	Binary Verification	2 py

(e) Binary Verification Effort

	Total Effort	Artefacts	Effort
CapDL Effort	2 py *	capDL Spec	0.6 py (7.2 pm) *
		capDL-to-Abstract Spec refinement proof	1.4 py (17.2 pm) *

(f) capDL Effort

Fig. 19. Development and Proof Efforts (starred numbers come from rigorous effort counting, others are conservative estimates).

per week. These are indicated with a star in the table. The nonstarred effort numbers are rough, conservative estimates. It should be noted that some starred numbers, for example, for the integrity and confidentiality proofs, improve upon the previously reported conservative estimates for these tasks [Sewell et al. 2011; Murray et al. 2013].

7.2.2. Kernel Development Effort. As shown in Figure 19(a), about 2 py went into the Haskell prototype (over all project phases), including design, documentation, coding, and testing. The initial C translation was done in 3 weeks, with a total of about 2 pm for the C implementation. This results in a total effort of 2.2 py for the whole seL4 microkernel development. 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 [Dannowski 2009].

SLOCCount [Wheeler 2001] 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.

7.2.3. Correctness Proof Effort. The effort for proving the correctness of seL4 is significantly higher than developing the kernel in the first place, in total about 20 py, as shown in Figure 19(b). The abstract specification took about 4 pm to develop. The executable specification only required setting up the translator; this took 3 pm. The total of 20 py for the proofs 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 replicating 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. Our total cost per SLOC is \$362 for the whole 22.7 py; it would be \$127/SLOC for the estimated 8 py involved in replicating it for a new kernel. This is low cost for an assurance level that is higher than what is required for EAL7, which is the Common Criteria's highest level of certification. EAL7 does not require the code to be formally linked to the design model, and the industry rule-of-thumb for the lower EAL6 is already \$1 k/LOC.[3]

The breakdown of effort between the two refinement stages is illuminating: The first refinement step (from abstract to executable specification) consumed 8 py, the second (to C semantics) less than 3 py, almost a 3:1 breakdown. This is a reflection of the (intentionally) 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 specification 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 specification layer to substantiate optimisations in C.

7.2.4. Optimisation Proof Effort. The largest performance optimisation that has been included in the verified version of the kernel is the fastpath, described in Section 4.6. The effort was about 5 pm, including updates to the second refinement as well as verification

[3]Klein et al. [2009b] contained an embarrassing typo, claiming \$10 k/LOC.

of the executable specification of the fastpath. This verification was conducted by one experienced engineer working part time on this project over 7 months. While that number may appear large, the optimisation in question is substantial and required developing new extensions to the verification framework.

Other, small-scale local optimisations that took about 1 day to implement took about 2–3 days to verify for one person. Verification does not always need to lead to more effort. In one case, knowledge from invariants in the verification reduced an optimisation patch from about 20 source lines to just a single line.

7.2.5. Security Proof Effort. A major benefit of the functional correctness proof is the greatly reduced effort required for code-level security proofs (see Figure 19(d), also highlighted in the proof sizes difference in Figure 17(b)). The strength of the refinement proof ensures that any property that is preserved by refinement can now be proved about the abstract, much simpler model, and is then guaranteed to hold for the kernel source code with no additional effort.

This effort reduction is illuminating for the integrity and authority confinement proofs: both are naturally expressible as Hoare triples and therefore directly preserved by refinement. The proof can thus simply target the abstract specification from the functional correctness proof, representing an effort of merely 7.4 pm, an order of magnitude less than what we estimate it would have required if done directly on the source code level.

The confidentiality statement is not preserved by refinement without additional conditions. As we explained in Section 5.3, in order to be able to prove this statement only at an abstract level while keeping the code-level guarantee, we had to make the abstract specification deterministic, and thus update the functional correctness proofs to hold for this new abstract specification. This took about 18.5 pm. We also had to modify the scheduler to avoid a global transitive channel for information flow via scheduler decisions, which took about 1.8 pm. The proofs of confidentiality themselves took about 20.4 pm, leading to a total effort of 40.7 pm to have code-level proof that seL4 enforces confidentiality. While this represents about 5 times the effort of proving integrity and authority confinement, it is massively smaller than what proving it directly on the C implementation would have cost.

Overall, the security proofs represent only $78/SLOC, a very low figure for the strongest assurance ever produced about the security of a general-purpose OS kernel.

7.2.6. Binary Verification Effort. Computing the effort for the binary verification is less precise as it spanned 20 months and involved two people from different institutions, both contributing to other activities in parallel. The total effort is roughly 2 py, one person contributing 1.5 py, and the other 0.5 py.

7.2.7. CapDL Refinement Effort. The capDL specification of seL4 required an effort of 7.2 pm, involving three main contributors. The refinement proof between the capDL model of seL4 and its abstract specification was conducted in about 9 months for a total effort of 17.2 pm. It involved a team of four to five people, with one main contributor providing 9.7 pm alone.

7.3. Bugs Found

7.3.1. Defects Revealed and Changes Required by the Correctness Proof. Figure 20 summarises the changes to the code and models that were required by the refinement proofs.

The first refinement step led to some 300 changes in the abstract specification and 200 in the executable specification. About 50% of these changes relate to bugs in the associated algorithms or design, the rest were introduced for verification convenience.

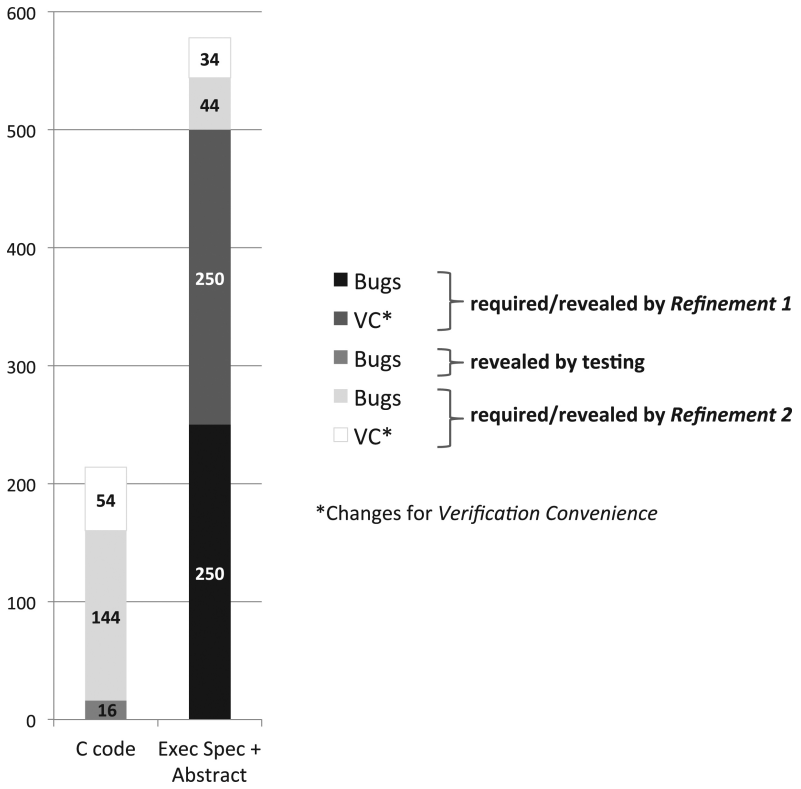


Fig. 20. Number of changes on the codes and models required by the verification.

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 Section 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 overly strong assumptions about what is true during execution. The bugs discovered in the second proof from executable specification to C were mainly typos, misreading the specification, or failing to update all relevant code parts on 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 could expect testing would identify. 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.

7.3.2. Defects Revealed and Changes Required by the Security Proof. The security proof uncovered many channels in the kernel, some of which were initially surprising even to those who had worked with seL4 for years. However, these channels do not represent code defects as such. Rather, they make explicit the requirements that must be met when using the general purpose seL4 kernel as a separation kernel: namely, that the system must be configured to deny these channels; the proof assumptions encode exactly how this configuration should be performed.

Apart from one minor change to simplify verification, the partition scheduler was the only change required to the seL4 C code. We address other channels (e.g., resulting from seL4's object deletion semantics, or seL4's interrupt delivery mechanism) by assuming explicit well-formedness conditions on the access-control policy and the initial configuration. These restrictions are common in high-assurance separation kernels.

Ultimately, our proof of information-flow security for seL4 makes seL4 no more secure than it was to begin with (excepting the implementation changes mentioned above). However, it provides a strong piece of *evidence* about the security of seL4 and its suitability as a separation kernel.

7.3.3. Defects Revealed and Changes Required by the Binary Verification. We did not find any genuine compiler flaws during this analysis. The seL4 team has reported experience with compiler defects in the past and fixed all known issues by rearranging code, selecting appropriate compiler versions and disabling some compiler flags, such as -fwhole-program. We did, however, find a number of small mismatches between our C semantics and the compiler's. None of them were serious, in the sense that they did not lead to incorrect behaviour, but their removal required some effort. This includes the treatment of the strict-aliasing rule and the handling of reserved sections (see Sewell et al. [2013] for more details).

7.4. The Cost of Change

An obvious issue of verification is the cost of proof maintenance: how much does it cost to reverify after changes are made to the kernel? This clearly 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 verification approach 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 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 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 0.25 pm.

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 pm 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 reverification cost about 32% of the time previously invested in verification.

The new features required only minor adjustments of existing invariants, but led 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 reproved 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 reverify.

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

8. RELATED WORK

We briefly summarise the literature on OS verification. Klein [2009] provides a comprehensive overview.

The first serious attempts to verify an OS kernel were in the late 1970s UCLA Secure Unix [Walker et al. 1980] and the Provably Secure Operating System (PSOS) [Feiertag and Neumann 1979]. 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 the specification and 20% of the implementation proofs in 5 py. The team concluded that invariant reasoning dominated the proof effort, which we found confirmed in our project.

PSOS was mainly focused on formal kernel design and never completed any substantial implementation proofs. Its design methodology was later used for the Kernelized Secure Operating System (KSOS) [Perrine et al. 1984] by Ford Aerospace. The Secure Ada Target (SAT) [Haigh and Young 1987] and the Logical Coprocessor Kernel (LOCK) [Saydjari et al. 1987] 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 [Walker et al. 1980]. 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 [Bevier 1989].

Bevier and Smith [1993] later produced a formalisation of the Mach microkernel without implementation proofs. Other formal modelling and proofs for OS kernels that

did not proceed to the implementation level include the EROS kernel [Shapiro et al. 1996], the high-level analysis of SELinux [Archer et al. 2003; Guttman et al. 2005] based on FLASK [Spencer et al. 1999], and the MASK project [Martin et al. 2000, 2002] which was geared towards information-flow properties. The MASK property resembles traditional unwinding conditions for noninterference, and was shown to hold for a low-level design model that is close to an implementation. Again, it was ultimately connected to the C implementation only by manual translation. Like many of the other kernels summarised here, MASK is not a general-purpose kernel as seL4, but instead designed primarily to enforce static separation. This means that the verification of seL4 is more complex and, at the same time, that more flexible kernel services are available inside separated seL4 partitions.

The VFiasco project [Hohmuth and Tews 2005] and later the Robin project [Tews et al. 2008] 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. [2006, 2008] report on the verification and Common Criteria certification of a "software-based embedded device" featuring a small (3,000 LOC) separation kernel. They focus on data separation rather than functional correctness. Their proof is machine-checked against an abstraction of their kernel, which they relate to their implementation with a pen-and-paper argument. Their formulation of separation involves a number of different properties: no exfiltration, no infiltration, temporal separation, control separation and kernel integrity. We can derive analogues for each of these properties for seL4 from our proof of information-flow security.

Krohn and Tromer [2009] presented a pen-and-paper proof of noninterference for the Flume operating system. This proof applied to a very abstract CSP [Hoare 1985] model of the Flume system, unconnected to its implementation by proof. Unlike seL4, Flume is Linux-based and so includes the entire Linux kernel as part of its trusted computing base.

Barthe et al. [2011] presented a formalisation and machine-checked proof of isolation for a high-level, idealised model of a hypervisor. More recent work in this vein [Barthe et al. 2012] has also looked at analysing cache leakage, which our proof does not, but again only for an idealised hypervisor model.

Hardin et al. [2006] formally verified information-flow properties of the AAMP7 microprocessor, 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 shown for the Green Hills Integrity kernel [Greenhills Software, Inc. 2008] during a Common Criteria EAL6+ evaluation [Richards 2010]. The Separation Kernel Protection Profile [Information Assurance Directorate 2007] of Common Criteria shows data separation only, and the proof applies to a handwritten, detailed formal model of the kernel that is not linked to its implementation by formal proof but instead by careful informal argument. This model likely has sufficient detail to cover all interesting code behaviours, but the missing formal and automatic connection to the code, let alone the binary, leaves open the possibility of implementation errors in INTEGRITY-178B that invalidate the proof of isolation. It also risks that the proof is not adequately updated when code or API change. The isolation proved for INTEGRITY-178B is based on the GWVr2 property [Greve 2010], which bears similarities to our formulation of information-flow security for seL4.

A number of kernels are designed specifically to enforce information-flow control, such as HiStar [Zeldovich et al. 2011] whose size is comparable to seL4's. HiStar

implements a simple semantics for enforcing information-flow control, based on object labels and category ownership. However, it does not have a formal proof that these rules correctly model the behaviour of the HiStar implementation, nor a formal connection between these rules and a high-level security property like noninterference.

The SAFE project [DeHon et al. 2011] follows a clean-slate approach to information-flow security, which includes new hardware design from scratch. While recent outcomes include interesting results for information-flow noninterference, such as new exception mechanisms [Hritcu et al. 2013], seL4 is a general-purpose kernel written in standard C, applicable to commodity platforms such as ARM and x86.

A closely related project in functional correctness is Verisoft [Alkassar et al. 2009, 2010c], which spanned not only the OS, but a whole software stack from verified hardware up to verified application programs. This includes a formally verified, nonoptimising compiler for their own Pascal-like implementation language. Even if not all proofs were completed, 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. Additionally, we cover high-level security properties that are generally applicable and directly useful for the verification of user-level systems.

The successor project, Verisoft-XT, initially attempted to verify functional correctness of the Microsoft hypervisor Hyper-V [Leinenbach and Santen 2009], but ultimately abandoned this proof in favour of demonstrating some of the relevant techniques on a small, idealised baby-hypervisor [Alkassar et al. 2010b]. The project shows significant depth in other aspects, most notably in dealing with multicore concurrency and the integration of weak memory models into the verification [Cohen and Schirmer 2010], as well as a deep treatment of virtual memory and shadow page table algorithms that are important for hypervisors [Alkassar et al. 2010a; Kovalev 2013]. The verified ARM version of seL4 uses para-virtualisation and can thereby avoid shadow page tables.

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 [Ball and Rajamani 2001] and BLAST [Henzinger et al. 2003]. They can show specific safety properties of C programs automatically, such as correct API usage in device drivers. The terminator tool [Cook et al. 2007] 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.

The Verve project [Yang and Hawblitzel 2010] proved correctness of a minimal language runtime, including garbage collection, to achieve provable type and memory safety on the assembly level for the kernel and its applications, which must run in the same language framework. While the functional correctness of the language runtime is a strong statement, type safety alone for the rest of the system is a weaker property compared to the security and functional correctness statements we have proved for seL4. Standard type safety on its own would for instance not support further access-control proofs about the kernel.

Implementations of other kernels in type-safe languages such as SPIN [Bershad et al. 1995] and Singularity [Fähndrich et al. 2006] 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.

Few of the kernel verification efforts above touch correct initialisation of user-level systems. The approach in EROS [Shapiro and Weber 2000] is to create and check the initial configuration manually, and for the rest of the lifetime of the system use persistent checkpoints instead of rebooting from scratch. On a higher level, SELinux introduces a policy language for configuring the access-control state of the system. These are large static policies on the level of files and applications, as opposed to the fine-grained kernel-level access-control mechanism that capDL describes. Hicks et al. [2007] developed a formal semantics for SELinux policies in Prolog and demonstrated that it was possible to show information flow properties of SELinux policies. Linux is of course too large a trusted computing base to formally verify that the corresponding access-control state is indeed achieved, let alone enforced.

The OKL4 microkernel [Open Kernel Labs 2008] moves the initialisation problem almost entirely to offline processing and runs the initialisation phase once, before the system image is built. Similarly to EROS, when the machine starts, it loads a fully pre-initialised state. While this makes it possible to inspect the initialised state offline, a full low-level assurance case must be made for each system. In our approach, assurance about system initialisation is now reduced to reasoning about static, formal capDL system descriptions.

Over the last 10+ years, a number of studies attempted to analyse the timeliness of real-time OSes, a summary is provided by Lv et al. [2009b]. The earliest work analysed the RTEMS OS [Colin and Puaut 2001]. It required significant manual intervention to deal with unstructured code. Furthermore, as the analysis was performed on the source-code level, it is necessarily pessimistic and must make assumptions about code generation.

A WCET analysis of the microkernel used by the OSE delta operating system was undertaken by Carlsson et al. [2002] and Sandell et al. [2004]. The OSE kernel is mostly preemptible, except for 612 short and mostly simple "disable-interrupt" regions, which the authors analysed. Lv et al. [2009a] analysed the WCET of each system call of the μC/OS-II kernel. The analysis was generally successful, although required a significant amount of manual intervention.

All these previously analysed systems are unprotected, single-mode kernels. Singal and Petters [2007] attempted an analysis on the L4 Pistachio kernel, a predecessor of seL4, which supports dual-mode execution and virtual memory. Unstructured code, inline assembly and context switching contributed to making the analysis difficult. Safe WCET bounds were never established.

9. CONCLUSIONS

The seL4 project represents the first comprehensive verification of an entire general-purpose OS kernel, providing a complete proof chain from the usual, high-level security and safety requirements, that is, integrity, confidentiality and availability, down to the executable machine code (a few remaining details notwithstanding).

On the way there it achieved a number of other firsts: the functional-correctness proof of a complete OS kernel implementation, the proof of its correct translation from C to binary, the proof of high-level security theorems such as integrity and intransitive noninterference for a general-purpose kernel implementation, and the sound

and complete analysis of worst-case execution times for an OS supporting full virtual memory.

All this was achieved on a system that is designed for real-world use: we did not trade average-case performance for verification (although we have potentially traded off worst-case performance, but this is difficult to ascertain in the absence of any comparable systems with similar timing analysis). Furthermore, seL4's design and implementation is based on our experience with earlier systems, including a predecessor which has been deployed in billions of devices, so we are confident that seL4 is similarly suitable for real-world deployment.

We have described the approach taken to achieve these outcomes, including a rapid prototyping methodology for kernel design, and design principles aiding verification, such as an event-based, mostly nonpreemptable kernel.

The possibly most encouraging lesson from the project, other than that it succeeded at all, is the cost: our analysis of the development and verification effort shows that, with the right approach, formally verified software is actually less expensive than traditionally engineered "high-assurance" software, yet provides much stronger assurance than what is possible with traditional software-engineering approaches.

Furthermore, our analysis indicates that the cost is only about a factor of two higher than that of software engineered to industry-standard (low-assurance) quality-assurance levels. This opens an exciting prospect for the future: should we be able to reduce the verification cost by another factor of two, then verified software will become cost-competitive under all circumstances, at least for the scale of systems we have been looking at.

We are optimistic that increased automation will achieve this in the near future.

ACKNOWLEDGMENTS

We would like to acknowledge the contribution of the following people in the different parts of this work, spanning multiple years and projects. Andrew Boyton, David Cock, Callum Bannister, Nelson Billing, Bernard Blackham, Timothy Bourke, Matthew Brassil, Adrian Danis, Matthias Daum, Jeremy Dawson, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Matthew Fernandez, Peter Gammie, Xin Gao, David Greenaway, Ihor Kuz, Corey Lewis, Daniel Matichuk, Jia Meng, Catherine Menon, Magnus Myreen, Michael Norrish, Sean Seefried, Yao Shi, David Tsai, Harvey Tuch, Adam Walker, and Simon Winwood.

REFERENCES

M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. 1986. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer USENIX Technical Conference*. USENIX Association, 93–112.

E. Alkassar, E. Cohen, M. A. Hillebrand, M. Kovalev, and W. Paul. 2010a. Verifying shadow page table algorithms. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design*. R. Bloem and N. Sharygina, Eds., IEEE, 267–270.

E. Alkassar, M. Hillebrand, D. Leinenbach, N. Schirmer, A. Starostin, and A. Tsyban. 2009. Balancing the load—leveraging a semantics stack for systems verification. *J. Automat. Reason.* Special Issue on Operating System Verification, 42, 2–4, 389–454.

E. Alkassar, M. Hillebrand, W. Paul, and E. Petrova. 2010b. Automated verification of a small hypervisor. In *Proceedings of Verified Software: Theories, Tools and Experiments*. G. Leavens, P. O'Hearn, and S. Rajamani, Eds., Lecture Notes in Computer Science, vol. 6217, Springer, 40–54.

E. Alkassar, W. Paul, A. Starostin, and A. Tsyban. 2010c. Pervasive verification of an OS microkernel: Inline assembly, memory consumption, concurrent devices. In *Proceedings of Verified Software: Theories, Tools and Experiments*. P. O'Hearn, G. T. Leavens, and S. Rajamani, Eds., Lecture Notes in Computer Science, vol. 6217, Springer, 71–85.

E. Alkassar, N. Schirmer, and A. Starostin. 2008. Formal pervasive verification of a paging mechanism. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. C. R. Ramakrishnan and J. Rehof, Eds., Lecture Notes in Computer Science, vol. 4963, Springer, 109–123.

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

J. Andronick, D. Greenaway, and K. Elphinstone. 2010. Towards proving security in the presence of large untrusted components. In *Proceedings of the 5th Systems Software Verification*. G. Klein, R. Huuck, and B. Schlich, Eds., USENIX, Berkeley, CA.

J. Andronick, R. Jeffery, G. Klein, R. Kolanski, M. Staples, H. J. Zhang, and L. Zhu. 2012. Large-scale formal verification in practice: A process perspective. In *Proceedings of the International Conference on Software Engineering*. ACM, 1002–1011.

M. Archer, E. Leonard, and M. Pradella. 2003. Analyzing security-enhanced Linux policy specifications. In *Proceedings of the 4th IEEE Workshop on Policies for Distributed Systems and Networks (POLICY)*. IEEE, 158–169.

ARM Ltd. 2005. *ARM1136JF-S and ARM1136J-S Technical Reference Manual.* R1P1 Ed., ARM Ltd.

T. Ball and S. K. Rajamani. 2001. SLIC: A specification language for interface checking. Tech. Rep. MSR-TR-2001-21, Microsoft Research.

J. Barhorst, T. Belote, P. Binns, J. Hoffman, J. Paunicka, P. Sarathy, J. Scoredos, P. Stanfill, D. Stuart, and R. Urzi. 2009. A research agenda for mixed-criticality systems. http://www.cse.wustl.edu/∼cdgill/CPSWEEK09_MCAR/.

G. Barthe, G. Betarte, J. D. Campo, and C. Luna. 2011. Formally verifying isolation and availability in an idealized model of virtualization. In *Proceedings of the 17th International Symposium on Formal Methods (FM)*. M. Butler and W. Schulte, Eds., Lecture Notes in Computer Science, vol. 6664, Springer, 231–245.

G. Barthe, G. Betarte, J. D. Campo, and C. Luna. 2012. Cache-leakage resilient OS isolation in an idealized model of virtualization. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium*. 186–197.

D. Bell and L. LaPadula. 1976. Secure computer system: Unified exposition and Multics interpretation. Tech. Rep. MTR-2997, MITRE Corp.

S. Berghofer. 2003. Proofs, programs and executable specifications in higher order logic. Ph.D. thesis, Institut für Informatik, Technische Universität München.

B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. 1995. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. ACM, 267–284.

W. R. Bevier. 1989. Kit: A study in operating system verification. *IEEE Trans. Soft. Eng.* 15, 11, 1382–1396.

W. R. Bevier and L. Smith. 1993. A mathematical model of the Mach kernel: Atomic actions and locks. Tech. Rep. 89, Computational Logic Inc. Apr.

M. Bishop. 2003. *Computer Security: Art and Science*. Addison-Wesley.

B. Blackham and G. Heiser. 2012. Correct, fast, maintainable – choose any three! In *Proceedings of the 3rd Asia-Pacific Workshop on Systems (APSys)*. 13:1–13:7.

B. Blackham and G. Heiser. 2013. Sequoll: A framework for model checking binaries. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*. Eduardo Tovar, Ed., IEEE, 97–106.

B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser. 2011. Timing analysis of a protected operating system kernel. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*. IEEE, 339–348.

B. Blackham, Y. Shi, and G. Heiser. 2012a. Improving interrupt response time in a verifiable protected microkernel. In *Proceedings of the 7th EuroSys Conference*. 323–336.

B. Blackham, V. Tang, and G. Heiser. 2012b. To preempt or not to preempt, that is the question. In *Proceedings of the 3rd Asia-Pacific Workshop on Systems (APSys)*. 8:1–8:7.

I. T. Bowman, R. C. Holt, and N. V. Brewster. 1999. Linux as a case study: Its extracted software architecture. In *Proceedings of the International Conference on Software Engineering*. 555–563.

A. Boyton, J. Andronick, C. Bannister, M. Fernandez, X. Gao, D. Greenaway, G. Klein, C. Lewis, and T. Sewell. 2013. Formally verified system initialisation. In *Proceedings of the 15th International Conference on Formal Engineering Methods*. Lindsay Groves and Jing Sun, Ed., Springer, 70–85.

P. Brinch Hansen. 1970. The nucleus of a multiprogramming operating system. *Communi. ACM* 13, 238–250.

M. Carlsson, J. Engblom, A. Ermedahl, J. Lindblad, and B. Lisper. 2002. Worst-case execution time analysis of disable interrupt regions in a commercial real-time operating system. In *Proceedings of the 2nd International Workshop on Real-Time Tools*.

D. Cock. 2008. Bitfields and tagged unions in C: Verification through automatic generation. In *Proceedings of the 5th International Verification Workshop*. B. Beckert and G. Klein, Eds., CEUR Workshop Proceedings, vol. 372, 44–55.

D. Cock, G. Klein, and T. Sewell. 2008. Secure microkernels, state monads and scalable refinement. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*. O. A. Mohamed, C. Muñoz, and S. Tahar, Eds., Lecture Notes in Computer Science, vol. 5170, Springer, 167–182.

E. Cohen and N. Schirmer. 2010. From total store order to sequential consistency: A practical reduction theorem. In *Proceedings of the 1st International Conference on Interactive Theorem Proving*. M. Kaufmann and L. Paulson, Eds., Lecture Notes in Computer Science, vol. 6172, Springer, 403–418.

A. Colin and I. Puaut. 2001. Worst case execution time analysis of the RTEMS real-time operating system. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*. 191–198.

B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. 2007. Proving that programs eventually do something good. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 265–276.

COYOTOS 2008. The Coyotos secure operating system. http://www.coyotos.org/.

J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. 2007. Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*. ACM, 351–366.

U. Dannowski. 2009. Personal communication.

M. Daum, N. Billing, and G. Klein. 2014. Concerned with the unprivileged: User programs in kernel refinement. *Form. Aspects Comput*. To appear.

L. M. de Moura and N. Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Lecture Notes in Computer Science, vol. 4963, Springer, Berlin, Germany, 337–340.

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

A. DeHon, B. Karel, B. Montagu, B. C. Pierce, M. Jonathan, F. Smithand Thomas, J. Knight, S. Ray, G. Sullivan, G. Malecha, G. Morrisett, R. Pollack, R. Morisset, and O. Shivers. 2011. Preliminary design of the SAFE platform. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems (PLOS)*.

J. B. Dennis and E. C. Van Horn. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM* 9, 143–155.

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

D. Elkaduwe. 2010. A principled approach to kernel memory management. Ph.D. thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia. http://ssrg.nicta.com.au/.

D. Elkaduwe, P. Derrin, and K. Elphinstone. 2008. Kernel design for isolation and assurance of physical memory. In *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*. ACM, 35–40.

K. Elphinstone and G. Heiser. 2013. From L3 to seL4 – what have we learnt in 20 years of L4 microkernels? In *Proceedings of the ACM Symposium on Operating Systems Principles*. ACM, 133–150.

K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser. 2007. Towards a practical, verified kernel. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*. 117–122.

M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. 2006. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of the 1st EuroSys Conference*. 177–190.

R. J. Feiertag and P. G. Neumann. 1979. The foundations of a provably secure operating system (PSOS). In *Proceedings of the National Computer Conference, AFIPS Conference Proceedings*. 329–334.

B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann. 1999. Interface and execution models in the Fluke kernel. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 101–115.

A. Fox. 2003. Formal specification and verification of ARM6. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*. D. Basin and B. Wolff, Eds., Lecture Notes in Computer Science, vol. 2758, Springer, 25–40.

A. Fox and M. Myreen. 2010. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Proceedings of the 1st International Conference on Interactive Theorem Proving*. M. Kaufmann and L. C. Paulson, Eds., Lecture Notes in Computer Science, vol. 6172, Springer, 243–258.

T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. 2003. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. ACM, 193–206.

J. Goguen and J. Meseguer. 1982. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 11–20.

M. J. C. Gordon, R. Milner, and C. P. Wadsworth. 1979. Edinburgh LCF. Lecture Notes in Computer Science, vol. 78, Springer.

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

D. A. Greve. 2010. Information security modeling and analysis. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*, D. S. Hardin, Ed., Springer, 249–300.

J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka. 2005. Verifying information flow goals in security-enhanced Linux. *J. Comput. Secur.* 13, 115–134.

J. T. Haigh and W. D. Young. 1987. Extending the noninterference version of MLS for SAT. *IEEE Trans. Softw. Engi.* 13, 141–150.

D. S. Hardin, E. W. Smith, and W. D. Young. 2006. A robust machine code proof framework for highly secure applications. In *Proceedings of the Workshop on the ACL2 Theorem Prover and its Applications.* 11–20.

G. Heiser. 2009. Hypervisors for consumer electronics. In *Proceedings of the 6th IEEE Consumer Communications and Networking Conference*. 1–5.

C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. McLean. 2006. Formal specification and verification of data separation in a separation kernel for an embedded system. In *Proceedings of the ACM Conference on Computer and Communications Security*. ACM, 346–355.

C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. McLean. 2008. Applying formal methods to a certifiably secure software system. *IEEE Trans. Softw. Engi.* 34, 1, 82–98.

T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. 2003. Software verification with Blast. In *Proceedings of the 10th SPIN Workshop on Model Checking Software*. Lecture Notes in Computer Science, vol. 2648, Springer, (Portland, OR). 235–239.

B. Hicks, S. Rueda, L. S. Clair, T. Jaeger, and P. D. McDaniel. 2007. A logical specification and analysis for SELinux MLS policy. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT)*. V. Lotz and B. M. Thuraisingham, Eds., ACM, 91–100.

C. A. R. Hoare. 1985. *Communicating Sequential Processes*. Prentice Hall.

M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro. 2004. Reducing TCB size by using untrusted components—small kernels versus virtual-machine monitors. In *Proceedings of the 11th SIGOPS European Workshop*, ACM.

M. Hohmuth and H. Tews. 2005. The VFiasco approach for a verified operating system. In *Proceedings of the 2nd Workshop on Programming Languages and Operating Systems (PLOS)*.

C. Hritcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. 2013. All your IFCException are belong to us. In *Proceedings of the IEEE Symposium on Security and Privacy.* IEEE, 3–17.

Information Assurance Directorate. 2007. U.S. government protection profile for separation kernels in environments requiring high robustness. Information Assurance Directorate, Version 1.03. http://www.niap-ccevs.org/cc-scheme/pp/pp.cfm/id/pp_skpp_hr_v1.03/.

G. Klein. 2009. Operating system verification—an overview. *Sādhanā* 34, 1, 27–69.

G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. 2010. seL4: Formal verification of an operating system kernel. *Communi. ACM* 53, 6, 107–115.

G. Klein, P. Derrin, and K. Elphinstone. 2009a. Experience report: seL4—formally verifying a high-performance microkernel. In *Proceedings of the 14th International Conference on Functional Programming*. ACM, 91–96.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. 2009b. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM Symposium on Operating Systems Principles*. ACM, 207–220.

G. Klein, T. Murray, P. Gammie, T. Sewell, and S. Winwood. 2011. Provable security: How feasible is it? In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems*. (Napa, CA). USENIX Association, 28–32.

R. Kolanski. 2011. Verification of programs in virtual memory using separation logic. Ph.D. thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia. http://ssrg.nicta.com.au/.

R. Kolanski and G. Klein. 2009. Types, maps and separation logic. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*. S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds., Lecture Notes in Computer Science, vol. 5674, Springer, 276–292.

M. Kovalev. 2013. TLB virtualization in the context of hypervisor verification. Ph.D. thesis, Saarland University, Saarbrücken, Germany.

M. Krohn and E. Tromer. 2009. Noninterference for a practical DIFC-based operating system. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 61–76.

I. Kuz, G. Klein, C. Lewis, and A. Walker. 2010. capDL: A language for describing capability-based systems. In *Proceedings of the 1st Asia-Pacific Workshop on Systems (APSys)*. 31–36.

L4HQ. 2007. http://l4hq.org/arch/arm/.

L4Ka Team. 2004. L4Ka::Pistachio kernel. http://l4ka.org/projects/pistachio/.

B. W. Lampson. 1971. Protection. In *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems*. Princeton University, 437–443.

D. Leinenbach and T. Santen. 2009. Verifying the Microsoft Hyper-V hypervisor with VCC. In *Proceedings of the 2nd World Congress on Formal Methods (FM)*. A. Cavalcanti and D. Dams, Eds., Lecture Notes in Computer Science, vol. 5850, Springer, 806–809.

X. Leroy. 2006. Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. J. G. Morrisett and S. L. P. Jones, Eds., ACM, (Charleston, SC). 42–54.

X. Leroy. 2012. Compcert version 1.10. http://compcert.inria.fr.

X. Li, Y. Liang, T. Mitra, and A. Roychoudhury. 2007. Chronos: A timing analyzer for embedded software. *Science Computer Program.* (Special issue on Experimental Software and Toolkit) 69, 1–3, 56–67.

Y.-T. Li, S. Malik, and A. Wolfe. 1995. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*. IEEE, 298–307.

J. Liedtke. 1993. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. ACM, 175–188.

J. Liedtke. 1996. Towards real microkernels. *Communi. ACM* 39, 9, 70–77.

J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, and T. Jaeger. 1997. Achieved IPC performance (still the foundation for extensibility). In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*. 28–31.

R. J. Lipton and L. Snyder. 1977. A linear time algorithm for deciding subject security. *J. ACM* 24, 3, 455–464.

M. Lv, N. Guan, Y. Zhang, R. Chen, Q. Deng, G. Yu, and W. Yi. 2009a. WCET analysis of the μC/OS-II real-time kernel. In *Proceedings of the 12th International Conference on Computational Science and Engineering*. 270–276.

M. Lv, N. Guan, Y. Zhang, Q. Deng, G. Yu, and J. Zhang. 2009b. A survey of WCET analysis of real-time operating systems. In *Proceedings of the 9th IEEE International Conference on Embedded Systems and Software*. IEEE, 65–72.

W. B. Martin, P. White, F. Taylor, and A. Goldberg. 2000. Formal construction of the mathematically analyzed separation kernel. In *Proceedings of the 15th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 133–141.

W. B. Martin, P. White, and F. S. Taylor. 2002. Creating high confidence in a separation kernel. *Automat. Softw. Engi.* 9, 3, 263–284.

D. Matichuk and T. Murray. 2012. Extensible specifications for automatic re-use of specifications and proofs. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*. 8.

T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. 2013. seL4: from general purpose to a proof of information flow enforcement. In *Proceedings of the Symposium on Security and Privacy*. IEEE, 415–429.

T. Murray, D. Matichuk, M. Brassil, P. Gammie, and G. Klein. 2012. Noninterference for operating system kernels. In *Proceedings of the 2nd International Conference on Certified Programs and Proofs*. Chris Hawblitzel and Dale Miller, Ed., Springer, 126–142.

M. O. Myreen. 2008. Formal verification of machine-code programs. Ph.D. thesis, University of Cambridge, Computer Laboratory, Cambridge, UK.

Z. Ni, D. Yu, and Z. Shao. 2007. Using XCAP to certify realistic system code: Machine context management. In *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics*. Lecture Notes in Computer Science, vol. 4732, Springer, 189–206.

NICTA. 2006. Iguana. http://www.ertos.nicta.com.au/software/kenge/iguana-project/latest/.

NICTA. 2013a. sel4 microkernel. http://ertos.nicta.com.au/research/sel4/.

NICTA. 2013b. Worst-case execution time computation tools. http://ssrg.nicta.com.au/software/TS/wcet-tools/.

T. Nipkow, L. Paulson, and M. Wenzel. 2002. Isabelle/HOL — A proof assistant for higher-order logic. In Lecture Notes in Computer Science, vol. 2283, Springer.

Open Kernel Labs. 2008. OKL4 web site. http://wiki.ok-labs.com/PreviousReleases.

J. Peleska, E. Vorobev, and F. Lapschies. 2011. Automated test case generation with SMT-solving and abstract interpretation. In *Proceedings of the NSAS Formal Methods Symposium*. Springer, 298–312.

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

QNX. 2012. Operating systems. http://www.qnx.com/products/neutrino-rtos/.

R. J. Richards. 2010. Modeling and security analysis of a commercial real-time operating system kernel. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*, D. S. Hardin, Ed., Springer, 301–322.

J. Rushby. 1992. Noninterference, transitivity, and channel-control security policies. Tech. Rep. CSL-92-02, SRI International.

J. M. Rushby. 1981. Design and verification of secure systems. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*. ACM, 12–21.

D. Sandell, A. Ermedahl, J. Gustafsson, and B. Lisper. 2004. Static timing analysis of real-time operating system code. In *Proceedings of the 1st International Symposium on Leveraging Applications of Formal Methods*.

O. Saydjari, J. Beckman, and J. Leaman. 1987. LOCKing computers securely. In *Proceedings of the National Computer Security Conference*. 129–141.

A. Seshadri, M. Luk, N. Qu, and A. Perrig. 2007. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. ACM, 335–350.

T. Sewell, M. Myreen, and G. Klein. 2013. Translation validation for a verified OS kernel. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. (Seattle, WA) ACM, 471–481.

T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein. 2011. seL4 enforces integrity. In *Proceedings of the 2nd International Conference on Interactive Theorem Proving*. M. C. J. D. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, Eds., Lecture Notes in Computer Science, vol. 6898, Springer, (The Netherlands). 325–340.

J. S. Shapiro, D. F. Faber, and J. M. Smith. 1996. State caching in the EROS kernel—implementing efficient orthogonal peristence in a pure capability system. In *Proceedings of the 5th IEEE International Workshop on Object Orientation in Operating Systems (IWOOOS)*. IEEE, 89–100.

J. S. Shapiro, J. M. Smith, and D. J. Farber. 1999. EROS: A fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*. ACM, 170–185.

J. S. Shapiro and S. Weber. 2000. Verifying the EROS confinement mechanism. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 166–181.

M. Singal and S. M. Petters. 2007. Issues in analysing L4 for its WCET. In *Proceedings of the 1st International Workshop on Microkernels for Embedded Systems (MIKES)*. NICTA.

L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. 2006. Reducing TCB complexity for security-sensitive applications: Three case studies. In *Proceedings of the 1st EuroSys Conference*. 161–174.

K. Slind and M. Norrish. 2008. A brief overview of HOL4. In *Proceedings of the Theorem Proving in Higher Order Logics, 20th International Conference*. Otmane Ait Mohamed, Csar Muoz, and Sofine Tahar, Ed., Springer, (Canada). 28–32.

R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. 1999. The Flask security architecture: System support for diverse security policies. In *Proceedings of the 8th USENIX Security Symposium*. USENIX Association. 123–139.

U. Steinberg and B. Kauer. 2010. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th EuroSys Conference*. 209–222.

H. Tews, T. Weber, and M. Völp. 2008. A formal model of memory peculiarities for the verification of low-level operating-system code. In *Proceedings of the 3rd Systems Software Verification*. R. Huuck, G. Klein, and B. Schlich, Eds., Electronic Notes in Theoretical Computer Science, vol. 217, Elsevier, 79–96.

H. Tuch. 2008. Formal memory models for verifying C systems code. Ph.D. thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia.

H. Tuch. 2009. Formal verification of C systems code: Structured types, separation logic and theorem proving. *J. Automat. Reason.* (Special Issue on Operating System Verification) 42, 2–4, 125–187.

H. Tuch, G. Klein, and G. Heiser. 2005. OS verification—now! In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*. 7–12.

H. Tuch, G. Klein, and M. Norrish. 2007. Types, bytes, and separation logic. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. M. Hofmann and M. Felleisen, Eds., ACM, 97–108.

D. von Oheimb. 2004. Information flow control revisited: Noninfluence = noninterference + nonleakage. In *Proceedings of the 9th European Symposium on Research in Computer Security*. P. Samarati, P. Ryan, D. Gollmann, and R. Molva, Eds., Lecture Notes in Computer Science, vol. 3193, Springer, 225–243.

M. von Tessin. 2010. Towards high-assurance multiprocessor virtualisation. In *Proceedings of the 6th International Verification Workshop*. M. Aderhold, S. Autexier, and H. Mantel, Eds., EasyChair Proceedings in Computing, vol. 3, EasyChair, 110–125.

M. von Tessin. 2012. The clustered multikernel: An approach to formal verification of multiprocessor OS kernels. In *Proceedings of the 2nd Workshop on Systems for Future Multi-Core Architectures*. Microsoft, 1–6.

M. von Tessin. 2013. The clustered multikernel: An approach to formal verification of multiprocessor operating-system kernels. Ph.D. thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia.

B. J. Walker, R. A. Kemmerer, and G. J. Popek. 1980. Specification and verification of the UCLA Unix security kernel. *Commun. ACM* 23, 2, 118–131.

D. A. Wheeler. 2001. SLOCCount. http://www.dwheeler.com/sloccount/.

A. Whitaker, M. Shaw, and S. D. Gribble. 2002. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, CA, 195–210.

R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. 2008. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.* 7, 3, 1–53.

S. Winwood, G. Klein, T. Sewell, J. Andronick, D. Cock, and M. Norrish. 2009. Mind the gap: A verification framework for low-level C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*. S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds., Lecture Notes in Computer Science, vol. 5674, Springer, 500–515.

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

J. Yang and C. Hawblitzel. 2010. Safe to the last instruction: automated verification of a type-safe operating system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 99–110.

N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. 2011. Making information flow explicit in HiStar. *Commun. ACM* 54, 11, 93–101.

Received August 2013; accepted September 2013