An Overview of the Verification of the seL4 Microkernel

Lukas Stevens

Seminar — Formal Proof in Mathematics and Computer Science

21st June 2018

Contents

1 What is the seL4 Microkernel?

1.1 A Brief History of Microkernels

Our interactions with computers are shaped by operating systems (OS) such as Linux, Windows and Android. Therefore, understanding the inner workings of operating systems is not only interesting, but also important. At the core of every OS resides the kernel [Lin]. The kernel is responsible for providing the necessary abstractions that are needed to run the rest of the OS and user facing applications. These abstractions include memory management, i.e. mapping of virtual addresses into physical memory, and scheduling of threads. In order to provide these services, the kernel is the first part of the OS loaded into memory at startup, and it retains complete control over the whole system after that. Owing to its critical nature, the kernel is usually loaded into a protected memory region, which can not be accessed by the rest of the OS or user applications. These non-privileged programs can only interact with the kernel via a predefined interface, using so called system calls. Since the kernel is the only program able to modify the protected memory region, only a bug in the kernel can lead to a crash of the whole system. Therefore, to guarantee successful execution, every program ultimately has to trust that the kernel behaves correctly.

Historically, many successful operating systems, e.g. Windows with MS-DOS, Linux, and BSD, employed a monolithic kernel architecture, which means that the entire OS with its components such as thread scheduling, network and device drivers, memory management, paging etc. is contained in the privileged kernel space. Naturally, this design gives a large surface for bugs to occur and potentially impedes modularity by allowing different subsystems of the kernel to be interdependent. Considering these shortcomings, it seems sensible to minimise the amount of software in the kernel. This idea eventually led to the emergence of microkernels, sometimes also written μ-kernel. Jochen Liedtke [Lie95], the inventor of the L4 microkernel, gives the following characterisation of a microkernel.

> A concept is tolerated inside the μ-kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality.

Allowing competing implementations of concepts like file systems and device drivers is advantageous as it improves the flexibility and extensibility of the system. However, there is no free lunch: the increased flexibility of microkernels came at a considerable performance penalty [Lie96]. This was especially noticeable in the first generation of microkernels, to which Chorus [Gui82], Mach [Gol90] and L3 [Lie93a] belong. Despite significant efforts, initial iterations of first generation microkernels still exhibited poor performance [Lie95]. Nevertheless, Liedtke was later able to eliminate major performance bottlenecks in a breakthrough result [Lie93b]. Subsequently, these improvements were incorporated into the L4 microkernel [Lie95] therefore marking the advent of a new generation of microkernels.

The original implementation of the L4 microkernel was later adapted multiple times. These different implementations are subsumed within the L4 microkernel family, an overview of which can be found elsewhere [EH13]. Later iterations of L4 provide various improvements over the original L4 kernel, e.g. performance improvements or additional security features, but they all honour the aforementioned design principle and therefore have a similar kernel interface. In particular, the kernel generally only provides the following hardware abstractions:

- virtual memory management, i.e. mapping virtual to physical addresses,
- a thread abstraction and scheduler, and
- an inter-process communication mechanism (IPC) that facilitates exchange of messages between different virtual address spaces.

Services like device and network drivers, file systems, and paging are not part of the kernel.

1.2 seL4: Putting the Security in L4

As the title suggests, we will consider one particular entry in the L4 microkernel family, namely the seL4 microkernel. Since seL4 is a microkernel, its code base is significantly smaller than that of a traditional monolithic kernel; compare seL4 with around 9,000 lines of C code [Kle+14] to the Linux kernel, which has over a million lines worth of code. Due to the relatively small size of microkernels, giving a formal specification of the kernel becomes tractable [TKH05; EH13]. By formal specification we mean an exhaustive description of the kernel's functional behaviour in mathematical terms. For example, the specification of seL4 states that each system call always terminates. Indeed, the seL4 kernel comes with a complete formal specification of its behaviour. Furthermore, it was proven that the implementation matches the specification using machine-checked proofs. Therefore, we acquire a certificate of the kernel's functional correctness with respect to its specification. Barring hardware defects, seL4 is the first general-purpose OS kernel guaranteed to always behave correctly. Additionally, it also achieves high performance in comparison to other L4 systems and provides a capability based access control mechanism [Kle+09].

By eliminating the need to blindly trust the kernel, seL4 provides a platform for systems with high security and robustness requirements, e.g. drone operating systems [Datb]. Programs with high security and low security demands can also be run concurrently on seL4, since it is guaranteed that their operations do not interfere, assuming they are not communicating with each other using the kernel IPC [Kle+14].

In the following, we only present how the correctness of the kernel's behaviour, i.e. its functional correctness, was proved. Above that, necessary and sufficient security properties for the soundness of seL4's access control mechanism were also established [Kle+14]. But before we examine seL4's proof of correctness, we will first discuss past approaches to verification of OS kernels.

1.3 Operating System Verification: An Overview

In the 1980s, one of the first ventures into specification and verification of an OS kernel was undertaken at University of California (UCLA) with the Unix Security kernel [WKP80]. The feature set of the Unix Security kernel was similar to today's microkernels. The correctness proof was divided into several abstraction layers with Pascal code at the bottom and the kernel specification at the top. Since the proofs were carried out manually in a machine-readable format, their soundness could be automatically checked. By viewing an execution of the kernel as a sequence of state transitions, it became possible to reason about its behaviour. Specification of the kernel was almost completed, but reasoning about the implementation

proved to be difficult. In the end, the verification effort was abandoned, but not after several bugs had been found during the proof process.

Almost ten years later, Bevier [Bev87; Bev89] presented his work on KIT, short for kernel of isolated tasks. KIT only has very restricted functionality, even in comparison to microkernels, and therefore could be implemented in less than 1,000 lines of assembler code. Because of that, thorough verification of the kernel became possible and the proof was carried out in Boyer-Moore logic using the automated Boyer-Moore theorem prover [BM90]. For verification, a similar proof structure to that of the UCLA Unix Security kernel was employed.

Another ten years later, the early 2000s gave rise to a new OS verification effort in the VFiasco (verified Fiasco) project [HT05]. The aim of the project was to obtain a correctness proof of the Fiasco kernel, which, like seL4, belongs to the L4 microkernel family. Since Fiasco was implemented in C++, a specification of the semantics of C++, albeit incomplete, was necessary in order for a correctness proof to be carried out. This semantics was one of the major contributions of the VFiasco project. On the other hand, the VFiasco project never completed the verification of the implementation.

Around the same time, the EROS microkernel [SSF99] introduced a capability based access control system to enhance the security of the OS employing it. It uses the take-grant capability model [LS77] in order to enforce mandatory access control. The access control system of seL4 is largely inspired by that of EROS. Furthermore, an abstract specification of the EROS access control system was formally verified [SW00] in a pen and paper proof, though this proof was never extended to the implementation. In the successor of EROS, called Coyotos, an implementation proof was supposedly a goal [Kle09]; however, development of Coyotos seems to have stopped. [1]

Lastly, we consider Verisoft [HP08], a project funded by the German government, whose purpose is an OS where verification does not stop at the compiler, but continues all the way down to the hardware until gate level. By that, the verification effort is not only concerned with the kernel, but also the compiler, the behaviour of the CPU, and user applications, e.g. an email client. In this undertaking, substantial work was put towards a complete verification of the compiler of a custom C-like language called C0A. Akin to seL4, Isabelle/HOL [NPW02] was used in the verification in order to mechanically check proof soundness. Nevertheless, the verification of the system remains incomplete. Finally, we remark that this glance at the topic of operating system verification is superficial at best and refer to a survey [Kle09] for an in-depth discussion.

2 Engineering the seL4 Kernel

2.1 Design Process for Verification

The focus of this and the following sections is the design process of the kernel with respect to verification and the resulting structure of the correctness proof. The intricacies of the kernel application programming interface (API), which consists of only a few system calls, will not be addressed here. Instead, we refer to a detailed treatment by Elphinstone and Heiser [EH13]. Often, kernel design follows a bottom-up approach, where the kernel is designed with the

[1] The website of Coyotos (`http://www.coyotos.org`) is unreachable at the time of writing.

details of the hardware and the corresponding low-level optimisations in mind. Accordingly, fundamental design issues may be discovered late, leading to costly rewrites of low-level, hardware dependent code. Naturally, this approach does not lend itself well to verification as verification generally follows a top-down approach. In verification, a top-down approach is beneficial since reasoning about concrete implementation details becomes tedious when proving abstract properties about a system. On the other hand, following this strategy harbours the risk of hiding important implementation details, thus leading to performance issues and hurting ease of implementation further down the line. Therefore, when using a top-down approach, it initially remains unclear whether it is possible to implement an usable system that fulfils the specification. Considering these shortcomings, the seL4 kernel was designed and implemented using what may be called a middle-out approach: at first, a prototype of the kernel in a high-level language was developed, which then inspired a formal specification of the kernel. An hardware-dependant low-level implementation of the kernel only followed as a last step.

As a first step, an Haskell prototype was implemented. Using Haskell has several advantages over low-level languages like C. Firstly, functions in Haskell are pure, meaning that they are stateless. Consequently, they behave much like mathematical functions, in that calling a function with the same arguments always yields the same result. Secondly, partly owing to the purity of Haskell functions, the semantics of Haskell is well-defined and simple compared to C. Thirdly, due to the similarity between Haskell and Isabelle/HOL, it is possible to automatically translate the Haskell implementation of the kernel into the language of the theorem prover, provided only a restricted subset of Haskell is used [Der+06; KDE09]. As a consequence, we obtain an executable specification of the kernel in the sense that the translated functions are still executable in Isabelle/HOL, but it also becomes possible to prove facts about them in Isabelle/HOL. Lastly, Haskell already has a mature ecosystem, which, among other things, made it possible to use the Haskell prototype as a hardware simulator [Kle+09], essentially turning it into a virtual machine. This step enables the kernel developers to test user applications against early versions of the kernel, which is crucial for rapid prototyping. Using the hardware simulator, the developers were able to test out different ideas and eventually change the requirements accordingly [Der+06]. As another advantage, no time must be spent to rewrite low-level, hardware dependent code when using a hardware simulation. But what about running the kernel on real hardware?

Unfortunately, interacting with the hardware directly, i.e. using hardware specific assembly instructions, is not possible in Haskell. Hence, the kernel must ultimately be implemented in a low-level language that facilitates interaction with the hardware and allows performance critical optimisations, which may otherwise violate Haskell's type safety. Specifically, the seL4 team decided to reimplement the kernel manually in C, thereby using the Haskell prototype as a blueprint. Additionally, there are further reasons that make it infeasible to use Haskell for a verified kernel: for one thing, Haskell needs a runtime, which is a larger body of code than the seL4 microkernel [Kle+09], and for another thing, Haskell relies on its runtime for garbage collection, which is unacceptable for a real-time OS, because garbage collection leads to unpredictable worst-case execution times. On the other hand, by virtue of using the prototype as a blueprint, some advantages of Haskell also translate over to the C implementation. For example, since Haskell functions are stateless, the kernel team was less likely to introduce unwanted side effects. In the end, the C implementation was swiftly completed in 2 person months [Kle+14]. Since the goal of the seL4 project is to obtain an usable and verified kernel, a correctness proof of the C implementation is mandatory.

To verify the C implementation, we first have to obtain some representation of it in our theorem prover. Therefore, a formalisation of the semantics of C was developed [Tuc09; Gre14]. This formalisation is used to automatically translate the code into a representation of its semantics in Isabelle/HOL [Tuc09; Gre14]. We then prove the correspondence between the semantic view of the C implementation and the executable specification, which is automatically generated from the Haskell prototype. This is considerably easier than linking the C implementation directly to the abstract specification, since the executable specification, in contrast to the abstract specification, already captures most implementation details. Finally, the relation between executable and abstract specification is established. Observe that this cuts the Haskell prototype from the proof chain, thus making it irrelevant for the verification effort (see Figure 1). However, one caveat is that the C compiler must be trusted to preserve the semantics of the above translation. In theory, we can circumvent this problem by also proving the correctness of the compiled binary.

2.2 Formal Methods of the Correctness Proof

As already mentioned, the seL4 project employs the theorem prover Isabelle/HOL [NPW02] to check proof soundness mechanically. However, this does not mean that the proofs are carried out automatically; on the contrary, Isabelle is an interactive theorem prover hence proofs have to be guided by human intervention. Before we dive deeper into the specifics of the verification effort, we first discuss the overarching concepts of the correctness proof. Specifically, functional correctness is formally shown using a proof principle called data refinement [DEB98]. A refinement proof establishes a correspondence between an abstract specification and a concrete, or refined, implementation of a system. Thereby, most properties of the abstract system are stated in terms of Hoare logic [Hoa69] or, more precisely, Hoare triples. An Hoare triple $\{P\}\, C\, \{Q\}$ encapsulates the following proposition: if P holds before the execution of the command C and C terminates, then Q holds after the execution of C. Consequently, $\{x = 1\}\, x := x + 1\, \{x = 2\}$ is an example for a valid Hoare triple if $:=$ is the assignment operator. Now, the pivotal property of data refinement is that valid Hoare triples in the abstract system are also valid in the concrete system, provided the latter refines the former. By transitivity, we conclude that properties of the abstract specification also hold for the low-level C implementation using the refinement chain (see Figure 1).

In Section 3, we will cover the layers of the functional correctness proof as seen in Figure 1. We start with the abstract specification, which describes the behaviour of the kernel rigorously, thereby fixing its interface. It does not, however, state how an implementation should achieve the specified behaviour. Secondly, we consider the translation of the Haskell prototype into Isabelle/HOL, namely the executable specification. Importantly, in contrast to the abstract specification, the executable specification already contains all data structures and algorithms relevant to the operation of the kernel. Below that comes the C implementation, which is essentially a translation of the Haskell prototype and thus uses almost the same data structures and algorithms. More precisely, the proof itself works with a representation of the semantics of the C code. Since this forces one to trust the C compiler, the C implementation was later eliminated from the proof chain by adding a verification layer for the compiled ARM binary. To this end, the kernel binary is imported into the theorem prover HOL4 and it is proved that it refines the semantics of the C code [Kle+14], but we will not discuss this step in further detail. But even with binary verification, there is no absolute verification: we still assume the correctness of the hardware and machine interface, and additionally trust

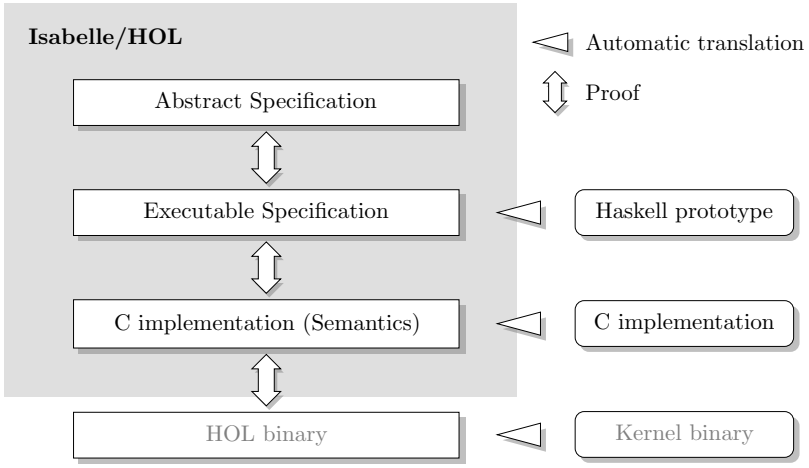

Figure 1: Layers of the functional correctness proof.

that the translation of the kernel binary to the HOL4 binary is correct. Lastly, we examine the refinement proofs between the abstraction layers in more detail.

3 Layers of the Functional Correctness Proof

3.1 Abstract Specification

The abstract specification is the most high-level layer still fully encapturing the functional behaviour of the kernel. Above that, there is an additional layer that concerns itself with the access control system of seL4 [Kle+14]. Due to its nature, the specification leaves as much implementation details unspecified as possible and only describes the expected outward behaviour of the kernel rather than how to achieve said behaviour. Therefore, the abstract specification necessarily fixes the kernel's API by predefining the argument formats of system calls up to their binary encoding as well as their error encodings. As a consequence, we achieve binary compatibility between all systems which refine the abstract specification. In other words, user applications will run on all refined systems if they are compatible with the abstract specification. To simplify verification, the use of data types with unrestricted size, for example natural numbers, is kept to a minimum. Instead, the specification primarily uses finite machine words, e.g. 32-bit integers. Memory and pointers are also modelled explicitly. Kernel data structures on the other hand are kept abstract in order to keep the implementation as unrestricted as possible. To this end, the kernel specification mainly uses sets, lists and trees as its data structures. Additionally, the abstract specification makes use of nondeterminism, that is an implementation is free to pick any option if there are multiple valid choices.

For an illustrative example, we inspect how the behaviour of the scheduler is defined on the abstract level as seen in Figure 2.[2] On the abstract level, no scheduling policy is prescribed

[2]Here, we present the example from the initial article about the verification of seL4 [Kle+09]; however, the implementation since has changed, though the general idea remains the same.

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

Figure 2: Isabelle/HOL implementation of the scheduler at the abstract level.

and thus the scheduler is specified as a function that picks any runnable active thread or the idle thread. The idle thread is a special thread, which is always runnable and whose purpose is to put the system into an idle state when there is no runnable thread. More concretely, the scheduler first retrieves the abstract set of all active runnable threads using `all_active_tcbs` and assigns it to `threads`. Then, the function `select` is used in order to obtain an arbitrary runnable thread. Finally, we are left with a nondeterministic choice, expressed by the `OR` syntax, whether to schedule `thread` or the idle thread.

3.2 Executable Specification

Descending in the proof hierarchy, we arrive at the executable specification, whose objective is to eliminate the nondeterminism of the abstract specification by giving a concrete implementation. Nevertheless, the executable specification benefited from the abstract specification, since carrying out the refinement proof gave immediate feedback about the correctness and security of the Haskell prototype [Kle+14]. While the executable specification already contains all the same kernel data structures and algorithms as the C implementation, it avoids the messy details of the C code such as low-level optimisations and pointers. However, it is important to not abuse Haskell types such that they are difficult to model in C, since verification would become considerably harder as a result. For example, capabilities, which are essentially authorisation tokens for system calls, are 64-bit integers in both the Haskell prototype and the C implementation, though this is only done to meet the same size constraints as the C implementation. The layout of the capability inside this limited size is not prescribed [Kle+09].

As mentioned earlier, the executable specification is generated directly from the Haskell prototype using an handwritten tool.[3] On the downside, the process of automatic translation puts some restrictions on the Haskell code. For a start, no extensive use of laziness and only restricted use of type classes is allowed [Der+06]. Furthermore, due to constraints of Isabelle/HOL, it is required that all Haskell functions terminate. For simplicity of verification, only simple recursion patterns were used in the Haskell functions. All in all, the restrictions above lead to easy, often even automatic, termination proofs. As a corollary, the termination of all seL4 API calls directly follows.

Coming back to the scheduler example, its Haskell implementation refines the abstract implementation in the following way: it explicitly searches for the runnable thread with the highest priority by using time slices and a priority-based round robin algorithm [Kle+14]. If no such thread is found, it schedules the idle thread.

[3]The tool can be found at `https://github.com/seL4/l4v/tree/master/tools/haskell-translator`.

3.3 Low-level Implementation in C

Following up on the executable specification, the C implementation fills in the remaining details. In order to reason about the C implementation, we first have to represent the semantics of the C code in Isabelle/HOL. Since the semantics of C is quite involved, this required significant formalisation effort. Initially, the C memory model was formalised by Tuch [TKN07; Tuc08; Tuc09]. Building thereupon, a C-to-Isabelle parser[4] with the purpose of translating the C code into the Simpl language was developed. Simpl [Sch06] is a generic Isabelle/HOL framework for reasoning about semantics of imperative languages. Utilising this framework, verification of general properties was made more convenient by automatically abstracting the Simpl representation of the C code into a high-level proof calculus with a tool called autocorres [Gre14]. In particular, this C semantics covers a large C99 subset [Kle+09], including machine dependent word sizes, type-unsafe pointer casting, and pointer arithmetic. From this representation in the theorem prover we automatically obtain proof obligations that assert the safety of each pointer access to global variables by using a verification condition generator. Thereby, we ensure the absence of null pointer dereferences and correct alignment of memory accesses, i.e. no out-of-bounds reads or writes happen. This guarantees that no corrupted memory is ever read and memory is never corrupted by a write. Most memory safety proof obligations can be discharged easily. Furthermore, additional proof obligations are generated to show conformity with the C99 standard [Kle+09].

Since the automatic translation only covers a subset C99, this leaves the question which parts of the C99 standard are not covered. The most momentous restriction is that obtaining references to local variables via the address-of operator is disallowed, which is frequently used to avoid passing large types by value. The purpose of this restriction is to strictly separate local from global variables and therefore simplify the generated proof obligations. Another advantage is that this restriction brings the C implementation closer to the Haskell prototype, because functions must be pure with exception of modifying the global state, which is modelled explicitly in Haskell. To circumvent performance bottlenecks caused by this restriction, reference parameters are avoided whenever possible and otherwise references to global variables are used instead. Other features excluded for verification convenience include function calls through function pointers, `goto` statements, and fall-through cases in `switch` constructs [Kle+09].

The implementation of a kernel sometimes requires direct interaction with the underlying hardware. Therefore, the C implementation makes use of inline assembly to accomplish tasks such as flushing the translation lookaside buffer (TLB). The effects of these assembler instructions, however, are not part of the aforementioned C semantics, because they are hardware dependent. Nevertheless, some hardware instructions, e.g. cache and TLB flushes, are relevant for the functional correctness of the kernel [Kle+14]. To effectively reason about these hardware instructions, they are hidden behind functions, whose effects on the machine state can be explicitly modelled on all higher abstraction layers. Consequently, we are now required to assure that the hardware instructions actually conform to their specification. In these limited cases where hardware instructions are used, the seL4 team relies on testing to check whether the C implementations of these functions behave as expected, yet their correctness has not been proven [Kle+14]. In the executable and abstract specification, these functions are once again maximally underspecified such that the proofs still work. In some

[4]The parser is located at `https://github.com/seL4/l4v/tree/master/tools/c-parser`.

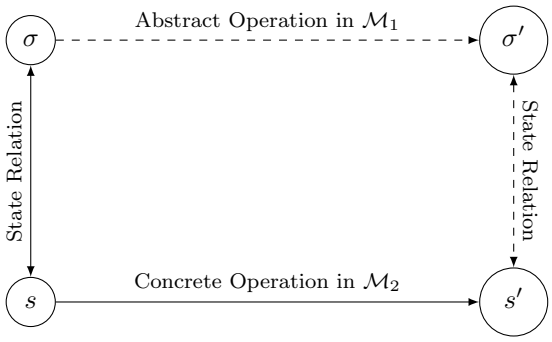

Figure 3: Forward simulation.

cases, an instruction may even arbitrarily change the state of the component it is responsible for. Leaving the behaviour underspecified makes the proofs more general, since we assume less about the hardware.

3.4 Refinement by Forward Simulation

To prove functional correctness with respect to the abstract specification, we need to establish the properties of the abstract system for its concrete implementations. In order to prove this, the system is seen as a state machine, where the functions that operate on the system state constitute state transitions. Consequently, abstract properties of the kernel are formulated as predicates on the state. Properties of the abstract specification A then extend to the concrete implementation C if program C refines A, i.e. the behaviour of C is contained in the behaviour of A. More precisely, seL4 uses the formal setting of data refinement [DEB98], where the system state is composed of the user-visible machine state and the kernel state. In this context, C must only refine the user-observable behaviour of A, while the internal kernel state can be modified arbitrarily. The seL4 team proved the properties of refinement for general state machines in Isabelle/HOL. The refinement proofs between the layers as seen Figure 1 are then just instantiations of this general framework [Kle+09].

So far, we are still forced to prove the refinement relation for arbitrary state transition sequences. To alleviate these limitations, refinement of an abstract state machine $\mathcal{M}_1$ by a concrete state machine $\mathcal{M}_2$ is established inductively using forward simulation, which is depicted in Figure 3. Initially, we assume that the concrete state s and the abstract state σ are contained in a relation R. Then, for any concrete state transition in $\mathcal{M}_2$ from the state s to the set of states s', there must exist a complementary state transition in $\mathcal{M}_1$ from σ to the set σ' such that s' and σ' are still related through R. Here, we have sets of resulting states because the transitions may be nondeterministic. In the end, this correspondence has to be shown for all possible transitions and a fixed relation R to arrive at refinement.

In the case of seL4 kernel, the general refinement framework is instantiated with a state machine with the following types of state transitions: kernel transitions, user transitions, user events, idle transitions, and idle events [Kle+09]. As Figure 4 illustrates, kernel transitions are atomic executions of the kernel. After their completion, control is transferred back to a runnable thread. If no runnable user thread exists, the system transitions into idle mode,

where it remains until an idle event, e.g. a timer interrupt, occurs. The above transitions are those modelled by the specification layers described by Figure 1. On the other hand, the user, modelled via user transitions, may haphazardly modify user-accessible parts of the state space. Additionally, the user can trigger a user event with system calls, which the kernel handles in a kernel transition.

The properties of the kernel and idle transitions are stated and proved in Hoare logic using Hoare triples. Usually, an Hoare triple $\{P\}\ C\ \{Q\}$ asserts partial correctness, i.e. that Q holds after the execution of C only if P holds in the beginning and, importantly, C terminates. Therefore, the termination of all kernel transitions is proved separately to obtain total correctness [Kle+09]. After termination, the next step is to prove invariants, which are properties that remain true throughout every execution. The invariants can be put into three broad categories [Kle+09]:

- low-level memory and typing invariants,
- data structure invariants, and
- algorithmic invariants.

Examples of the first category include that the kernel objects are aligned to their size and do not overlap, together with kernel objects having a well-defined type, and the references within them again pointing to objects of the correct type. Secondly, data structure invariants primarily assert that the kernel data structures are well-formed, that is, certain lists are always terminated or doubly linked lists have correct back links. The last category is concerned with the behaviour of the kernel, e.g. that the idle thread is always idle and the only thread which is idle, or admissibility of certain optimisations.

Since invariants hold for all states, they can subsequently be used in the forward simulation proofs to establish refinement. In summary, if $\mathcal{M}_A$, $\mathcal{M}_E$ and $\mathcal{M}_C$ denote the state machine instantiated with the abstract specification, the executable specification and the semantic representation of the C implementation respectively, then the following theorems are to be proved.

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

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

Thus, by transitivity of refinement, we get the following.

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

To no surprise, statements involving the kernel transitions require the most verification effort when proving these theorems. In addition to forward simulation, nonfailure of state transition is also proved, which shows that the kernel never crashes. Together with the invariants, whose proofs rely on nonfailure, the above theorems follow as a consequence. We will only sketch how these proofs are carried out; the formal details can be found elsewhere [CKS08; Win+09]. Generally, each kernel state transition corresponds to a single function in the specification [Kle+14]. Therefore, most C functions also have a counterpart in the executable and abstract specification. Due to the similar structure of functions in adjacent layers in Figure 1, correspondence of whole functions can be derived from the correspondence of related function fragments. For small enough function fragments a direct correspondence proof via Hoare triples becomes feasible. Thereby, the correspondence between abstract and executable

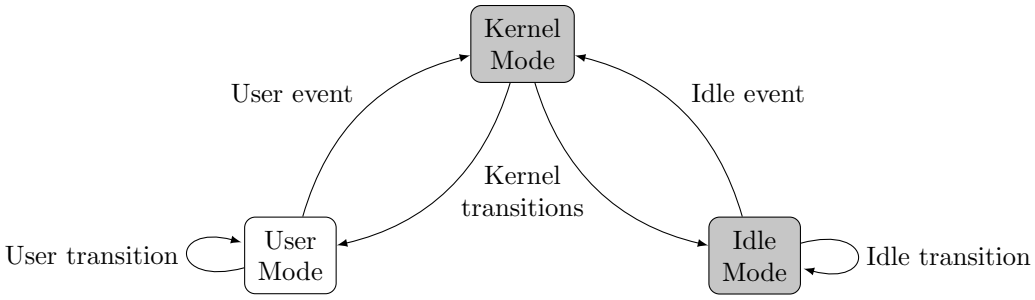

Figure 4: Types of transitions in seL4.

specification is typically easy to prove, which seems sensible considering that the abstract specification was primarily inspired by the executable specification. On the other hand, the correspondence between the executable specification and the C implementation is more challenging, owing largely to the pronounced difference of the C and the Haskell semantics. Nevertheless, the invariants were substantially more difficult to prove than refinement and constitute about 80% of the verification effort [Kle+09].

3.5 The Costs and Benefits of Verification

In the previous sections, we discussed the kernel's development and its proof of correctness. As artefacts of this endeavour, the seL4 team produced a C implementation of the kernel with around 9,000 software lines of code (SLOC) and an Haskell prototype with around 7,000 SLOC. Nevertheless, these artefacts are markedly dominated in size by the correctness proof, which takes up about 200,000 lines of proof script [Kle+14]. The seL4 kernel thus works as a clear example for the high price of verification. Therefore, when creating usable, high-assurance software, we have to weigh the benefits of formal verification against its inherently high cost. To this end, we will briefly summarise the experience of Klein et al. [Kle+14] with developing and verifying the seL4 microkernel. Table 1 gives an overview of the accumulated expenditure of time throughout different parts of the project. The expenditure of time is given in person years (py) and is a conservative estimate if it was not rigorously tracked.

At first, the kernel development team implemented the Haskell prototype, which required an effort of 2 py. This also includes the design and implementation of multiple kernel revisions as well as documentation and testing of the final version. As the C implementation is basically a translation of the Haskell prototype, it was promptly implemented in 3 weeks amounting to a total effort of 2 person months (pm). To put these numbers in perspective, the implementation of the Pistachio kernel, which also belongs to the L4 family, incurred an effort of about 6 py [Kle+14]. Therefore, prototyping the kernel in a higher level language first might turn out to be significantly more efficient even if verification is not the ultimate goal.

The proof effort, on the other hand, was considerably more time consuming than the implementation. A total of 20.5 py were spent in the different steps towards the complete verification of the kernel. Initially, 9.0 py were required to develop a collection of generic

	Artefact	Effort (py)	Total (py)
Kernel Development	Haskell prototype	2.0	2.2
	C implementation	0.2	
Correctness Proof	Generic framework and tools	9.0	20.5
	Abstract specification	0.3	
	Executable specification	0.2	
	Refinement $\mathcal{M}_A \leftrightarrow \mathcal{M}_E$	8.0	
	Refinement $\mathcal{M}_E \leftrightarrow \mathcal{M}_C$	3.0	
Binary Verification	Verified Binary	2.0	2.0

Table 1: Expenditure of time for seL4 development and proofs in person years (py).

frameworks and tools, to which, for example, the generic refinement and forward simulation framework belongs. After that, the abstract and executable specification could both be completed quickly since the abstract specification is essentially an abstraction of the Haskell prototype and the executable specification is automatically generated from the same prototype. Finally, the remaining time was spent to establish refinement between the different abstraction layers. The refinement proof between abstract and executable specification took 8.0 py, thereby dominating the effort of 3.0 py for the second refinement step. This is largely due to the fact that the abstract and executable specifications were developed in parallel and the actual refinement proof was only carried out after multiple design changes were applied. On the other hand, the second refinement step is challenging in principle since the semantics of C and Haskell are quite different, but a priori restrictions on the C coding style enforced a similar overall structure and therefore improved convenience of verification. Additionally, establishing invariants constituted about 80% of the verification effort; however, only few invariants were proved on the level of the executable specification, which explains why the first refinement required more work.

We established that constructing high-assurance platforms by means of formal verification is time consuming. This raises the question how the approach of seL4 compares to customary development paradigms in the industry. In particular, a frequently used certificate in the industry is the Evaluation Assurance Level 7 (EAL7), which is Common Criteria's[5] highest assurance standard. Still, EAL7 offers less extensive guarantees than full formal verification insofar that a formal model of the system is required, but it does not have to be linked to the implementation. Even for the weaker EAL6 certification, the industry rule-of-thumb [Kle+14] for cost is already $1,000/SLOC whereas the seL4 team estimates $362/SLOC over the whole 22.7 py of kernel development and verification. Therefore, using formal verification for designing high-assurance systems is not only feasible, but may also increase productivity.

The above shows that verification does incur considerable cost, but what is the payoff? In the case of seL4, the first refinement step induced 300 changes in the abstract and 200 changes in the executable specification respectively. Of these changes, 50% were due to bugs and the other half for verification convenience. Thereby, the bugs were mainly due to missing user input sanitation, subtle side effects intermittently breaking global invariants during

[5]Common Criteria is a standard for computer security certification defined by ISO/IEC 15408.

execution, or assuming unduly strong invariants. The second refinement step predominantly revealed typos, missing exception checking, and incorrect assumptions about default values. For example, the ARM interrupt controller returns 0xFF if no interrupt is active, but the C implementation checked against NULL at one point.

Finally, an integral part of verification is proof maintenance, or phrased differently, how many proofs break as a result of a change in the kernel. Naturally, this heavily depends on the type of change. The most favourable changes are optimisations as they do not change the behaviour and are therefore typically easy to verify. Another favourable change is adding new, independent features. For example, a new system call to batch together a short sequence of other system calls was implemented and verified in 0.25 pm. Adding large cross-cutting features, however, tends to be expensive. As an example, consider the addition of interrupts as well as ARM page tables and address spaces, which resulted in an re-verification effort of 1.5–2 py, while modifying 12% of the Haskell prototype. Lastly, fundamental changes to existing features, e.g. modifying the behaviour of system calls, should be avoided. In one case, a change to an existing system call took about 1 py to re-verify, even though it only modified about 5% of code base. In the end, we have to bear with these costs if we want to exclude the possibility of bugs through formal verification. On a positive note, no time is spent on implementation bug fixes after verification is complete.

4 Summary and Outlook

In this paper, we saw how the behaviour of the seL4 microkernel was verified with respect to its formal verification. Perhaps surprisingly, no disproportionate compromises in development time had to be made. Nevertheless, the verification proof does not come without limitations: we assume the correctness of the hardware, the machine interface, and the translation from the kernel binary into the HOL4 binary. As a general limitation, we also have to trust that the specification fully captures the desired behaviour of the kernel. In light of the recent Meltdown and Spectre hardware vulnerabilities, trusting the hardware may be ill-advised. For high-assurance systems, a comprehensive proof down to hardware level therefore becomes desirable. This, however, is not the focus of the seL4 project [Data]. Currently, work is put towards improving architecture and multicore support as well as adding new features. Specifically, kernel implementations for both the 32-bit and 64-bit versions of the x86 architecture exist, but their verification is still in progress. Similarly, multicore support is already present, yet verification is still pending. Lastly, implementation of strict temporal partitioning to exclude timing-channel attacks is also underway. All in all, the success of the seL4 project shows the practicality of provably secure systems and may inspire a move towards more verified software in the future.

References

[Bev87] William Bevier. 'A Verified Operating System Kernel'. PhD thesis. The University of Texas at Austin, 1987.

[Bev89] William Bevier. 'Kit: A study in operating system verification'. In: *IEEE Transactions on Software Engineering* (1989), pp. 1382–1396.

[BM90] Robert S. Boyer and J. Strother Moore. 'A theorem prover for a computational logic'. In: *Tenth International Conference on Automated Deduction*. Springer, 1990, pp. 1–15.

[CKS08] David Cock, Gerwin Klein and Thomas Sewell. 'Secure microkernels, state monads and scalable refinement'. In: *International Conference on Theorem Proving in Higher Order Logics*. Springer, 2008, pp. 167–182.

[Data] Data61. *Development and Verification Roadmap*. URL: `http://sel4.systems/Info/Roadmap/` (visited on 19/05/2018).

[Datb] Data61. *The SMACCM project*. URL: `http://ts.data61.csiro.au/projects/TS/SMACCM/` (visited on 01/05/2018).

[DEB98] Willem-Paul De Roever, Kai Engelhardt and Karl-Heinz Buth. *Data refinement: model-oriented proof methods and their comparison*. Cambridge University Press, 1998.

[Der+06] Philip Derrin et al. 'Running the Manual: An Approach to High-assurance Microkernel Development'. In: *Proceedings of the ACM SIGPLAN Workshop on Haskell*. ACM, 2006, pp. 60–71.

[EH13] Kevin Elphinstone and Gernot Heiser. 'From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels?' In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 133–150.

[Gol90] Golub, David and Dean, Randall and Forin, Alessandro and Rashid, Richard. 'UNIX as an Application Program'. In: *Procceedings of the Usenix Summer Conference*. Usenix Association, 1990, pp. 87–96.

[Gre14] David Greenaway. 'Automated proof-producing abstraction of C code'. PhD thesis. University of New South Wales, 2014.

[Gui82] Marc Guillemont. 'The Chorus distributed operating system: Design and implementation'. In: *Proceedings of the ACM International Symposium on Local Computer Networks*. Springer, 1982, pp. 207–223.

[HP08] Mark Hillebrand and Wolfgang Paul. 'On the Architecture of System Verification Environments'. In: *Hardware and Software: Verification and Testing*. Springer, 2008, pp. 153–168.

[Hoa69] C. A. R. Hoare. 'An Axiomatic Basis for Computer Programming'. In: *Communications of the ACM* (1969), pp. 576–580.

[HT05] Michael Hohmuth and Hendrik Tews. 'The VFiasco approach for a verified operating system'. In: *Second ECOOP Workshop on Programm Languages and Operating Systems* (2005).

[Kle09] Gerwin Klein. 'Operating system verification — An overview'. In: *Sadhana* (2009), pp. 27–69.

[KDE09] Gerwin Klein, Philip Derrin and Kevin Elphinstone. 'Experience Report: seL4: Formally Verifying a High-performance Microkernel'. In: *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2009, pp. 91–96.

[Kle+09] Gerwin Klein et al. 'seL4: Formal Verification of an OS Kernel'. In: *Proceedings of the ACM SIGOPS Twenty-Second Symposium on Operating Systems Principles*. ACM, 2009, pp. 207–220.

[Kle+14] Gerwin Klein et al. 'Comprehensive Formal Verification of an OS Microkernel'. In: *ACM Transactions on Computer Systems* (2014), 2:1–2:70.

[Lie93a] Jochen Liedtke. 'A persistent system in real use — experiences of the first 13 years'. In: *Proceedings of the Third International Workshop on Object Orientation in Operating Systems*. IEEE, 1993, pp. 2–11.

[Lie93b] Jochen Liedtke. 'Improving IPC by Kernel Design'. In: *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*. ACM, 1993, pp. 175–188.

[Lie95] Jochen Liedtke. 'On Micro-kernel Construction'. In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. ACM, 1995, pp. 237–250.

[Lie96] Jochen Liedtke. 'Toward Real Microkernels'. In: *Communications of the ACM* (1996), pp. 70–77.

[Lin] Linux Information Project. *Kernel Definition*. URL: `http://www.linfo.org/kernel.html` (visited on 28/04/2018).

[LS77] Richard Lipton and Lawrence Snyder. 'A Linear Time Algorithm for Deciding Subject Security'. In: *Journal of the ACM* (1977), pp. 455–464.

[NPW02] Tobias Nipkow, Lawrence C. Paulson and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS. Springer, 2002.

[Sch06] Norbert Schirmer. 'Verification of Sequential Imperative Programs in Isabelle/HOL'. PhD thesis. Technische Universität München, 2006.

[SSF99] Jonathan Shapiro, Jonathan Smith and David Farber. 'EROS: A Fast Capability System'. In: *Proceedings of the seventeenth ACM symposium on Operating systems principles* (1999), pp. 170–185.

[SW00] Jonathan Shapiro and Sam Weber. 'Verifying the EROS Confinement Mechanism'. In: *Proceedings of the IEEE Symposium on Security and Privacy* (2000), pp. 166–176.

[Tuc08] Harvey Tuch. 'Formal memory models for verifying C systems code'. PhD thesis. University of New South Wales, 2008.

[Tuc09] Harvey Tuch. 'Formal Verification of C Systems Code'. In: *Journal of Automated Reasoning* (2009), pp. 125–187.

[TKH05] Harvey Tuch, Gerwin Klein and Gernot Heiser. 'OS Verification — Now!' In: *Tenth Workshop on Hot Topics in Operating Systems*. Usenix Association, 2005, pp. 7–12.

[TKN07] Harvey Tuch, Gerwin Klein and Michael Norrish. 'Types, Bytes, and Separation Logic'. In: *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2007, pp. 97–108.

[WKP80] Bruce Walker, Richard Kemmerer and Gerald Popek. 'Specification and Verification of the UCLA Unix Security Kernel'. In: *Communications of the ACM* (1980), pp. 118–131.

[Win+09] Simon Winwood et al. 'Mind the Gap: A verification framework for low-level C'. In: *Theorem Proving in Higher Order Logics*. Springer, 2009, pp. 500–515.