Final Report AOARD 094160

Formal System Verification for Trustworthy Embedded Systems

June Andronick and Gerwin Klein

1 Introduction

This report summarises the work done in AOARD project 094160, Formal System Verification for Trustworthy Embedded Systems. We begin by revisiting the original motivation and work plan, continue with a brief high-level summary of the project outcomes and end with two publications [1, 2] that describe the outcomes in depth. We also attach two further papers, that were not directly funded by this project, but that give the necessary background definitions for this work [3, 4].

2 Motivation and work plan

The larger research topic was that of trustworthy embedded systems, in particular the formal verification of large, massively complex embedded systems. Our larger research vision is building provably secure systems on top of a formally verified microkernel basis. We had previously achieved code-level formal verification of the seL4 microkernel [3]. In the present project, over 12 months with 0.6 FTE, we concentrated on a verification framework that allows us to combine proofs from different levels of abstraction into one final, verifiable formal statement (e.g. combining MILS-style security architectures, component specifications, component code proofs, and OS kernel code proofs into a statement about access control in the overall system).

The large, 4-year research challenge is to integrate a number of differing formal analyses into one framework such that the overall effort for verification of the whole system is reduced to the following partial proofs: an automatic security analysis on an abstract system model, manual code proofs for a small number of component implementations, automatically generated

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **19 APR 2011**	2. REPORT TYPE **FInal**	3. DATES COVERED **24-09-2009 to 30-09-2010**

4. TITLE AND SUBTITLE **Formal System Verification for Trustworthy Embedded Systems**	5a. CONTRACT NUMBER **FA23860914160**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) **Gerwin Klein**	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of New South Wales,Locked Bag 6016,Sydney 1466,Australia,AU,1466**	8. PERFORMING ORGANIZATION REPORT NUMBER **N/A**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) **AOARD, UNIT 45002, APO, AP, 96337-5002**	10. SPONSOR/MONITOR’S ACRONYM(S) **AOARD**
	11. SPONSOR/MONITOR’S REPORT NUMBER(S) **AOARD-094160**

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report summarizes the work done in AOARD project 094160, Formal System Verification for Trustworthy Embedded Systems. It begins by revisiting the original motivation and work plan, continues with a brief high-level summary of the project outcomes, and ends with four publications that describe the background and outcomes in depth.

15. SUBJECT TERMS

Computational Logic, Software, Computer Operating Systems, Software Verification

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**40**	

glue code, and the previously formally verified microkernel. The framework will have to be general enough to encompass the different formalisms used in the different source proofs, but specific enough to make its application either low-effort or largely automatic.

The purpose of present project was to start the initial phase of this larger vision with a small case study, formalising an existing design that we can use to develop the full framework.

The work plan of the project had the following milestones.

- **Formal architecture specification:** The planned outcome of this milestone is an abstract, high-level formalisation of the case study design. The main intention of this formalisation is to capture which components exist in the system and how they are connected in terms of communication channels. The idea is that this formalisation will serve as an instance of a high-level security architecture specification to be connected to lower abstraction levels in the following milestones and that it is constructed with a view to be useful for a formal security analysis.

- **Formal component behaviour specification.** The planned outcome of this milestone is a formal behaviour specification of one or more trusted components in the case study system. The specification will assume communication primitives implemented either directly by microkernel calls or by generated communication stubs. The formalisation and verification of the implementation of these communication primitives is not part of this milestone (it is a later part of the larger project).

- **Formal system semantics.** The planned outcome of this milestone is a behaviour description for the whole system. The idea is to give a semantics for concurrently executing components that connects the previously constructed single component behaviour specifications with the architecture specification from the first milestone and pulls them together into a behaviour description of the whole system.

- **Formal analysis.** The planned outcome of this milestone is an investigation of how a simplified security analysis of the architecture specification (such as a simple connectedness analysis) can be mapped to the full formal behaviour specification of the system of the previous milestone. The main question to be answered is: which proof obligations need to be solved for the analysis of just the architecture to be meaningful for the next level down? Depending on the analysis, these obligations should be much simpler than a full formal refinement proof.

The main outcome we were aiming for in this one year was an initial framework prototype based on a small case study with support for describing a system on the abstract security architecture level, allowing the formal specification of concurrent behaviour of system components. The framework will not necessarily allow a connection to an existing microkernel proof yet or automatically generated configuration and glue code. The latter two are to be developed in later stages of the project.

3 Outcomes

The milestones in the previous section have all been achieved.

In a previous project, we designed and implemented a secure network access device (SAC) on top of the verified seL4 microkernel. The device allows a trusted front-end terminal to securely connect to separate back-end networks of different classification levels. On a switch request from the user via a web interface, the access device will connect the front-end terminal to the requested back-end network, one at a time. The security goal of the device is to prevent any information flows between back-end networks through the device.

For milestone 1, we have adapted and further developed a fully formal, high-level architecture specification of this SAC device. For milestone 2, we have formally described the behaviour of the so-called Router Manager component, the only trusted component in this system. For milestone 3, we have defined a high-level interleaving semantics for the execution of the whole system, including the kernel. For milestone 4, we have developed a formal security proof of the SAC that examines all possible executions of the high-level system.

All formalisations and proofs in this project have been conducted, and machine-checked in the theorem prover Isabelle/HOL. The security proof has additionally been modelled and checked in the the SPIN model checker.

The final deliverable of the project was a technical report or publication on the framework prototype described above. We have published two papers on the project. The first publication describes the formalisation and security proof in detail [1]. The second publication is an invited submission for the keynote presentation at the Asian Symposium of Programming Languages and Systems in 2010 [2]. It gives a detailed overview of the results, places them into the larger context, and outlines the work that still needs to be done to achieve an overall system security theorem.

References

[1] June Andronick, David Greenaway, and Kevin Elphinstone. Towards proving security in the presence of large untrusted components. In Gerwin Klein, Ralf Huuck, and Bastian Schlich, editors, *5th SSV*, Vancouver, Canada, Oct 2010. USENIX.

[2] Gerwin Klein. From a verified kernel towards verified systems. In Kazunori Ueda, editor, *8th APLAS*, volume 6461 of *LNCS*, pages 21–33, Shanghai, China, Nov 2010. Springer.

[3] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. *CACM*, 53(6):107–115, Jun 2010.

[4] Ihor Kuz, Gerwin Klein, Corey Lewis, and Adam Walker. capDL: A language for describing capability-based systems. In *1st APSys*, pages 31–36, New Delhi, India, Aug 2010.

Towards proving security in the presence of large untrusted components

June Andronick
NICTA, UNSW

David Greenaway
NICTA

Kevin Elphinstone
NICTA, UNSW

Abstract

This paper proposes a generalized framework to build large, complex systems where security guarantees can be given for the overall system's implementation. The work builds on the formally proven correct seL4 microkernel and on its fine-grained access control. This access control mechanism allows large untrusted components to be isolated in a way that prevents them from violating a defined security property, leaving only the trusted components to be formally verified. The first steps of the approach are illustrated by the formalisation of a multi-level secure access device and a proof in Isabelle/HOL that information cannot flow from one back-end network to another.

1 Introduction

Advances in machine-assisted theorem proving, and formal methods techniques in general, have pushed the limits of software verification to the point where it is possible to prove properties of real-world applications. The recently verified seL4 microkernel is one such example. Its 7500 lines of C code were formally proved to correctly implement a high-level abstract specification of its behaviour [7].

Formally verifying programs with sizes approaching 10 000 lines of code is a significant improvement in what formal methods was previously able to verify with reasonable effort. However, 10 000 lines of code is still a significant limit on the application of formal methods to the verification of contemporary software systems. Modern software systems, beyond very simple embedded systems, frequently consist of millions of lines of code. Thus the challenge remains as to how formal assurance can be given to real-world software systems of such size.

This paper presents our vision for how specifically targeted properties can be provably assured in very large and complex software systems. Our vision comes from the observation [1] that not all software in a large system necessarily contributes to a property of interest. For example, a game installed on a smartphone contributes nothing to the ability to make reliable phone calls. If one can assure the game is isolated from the phone call software, one can focus verification effort on the phone call software to assure reliability of calls.

The vision is to develop methodologies and tools that enable developers to systematically (*i*) isolate the software parts that are not critical to a targeted property, and prove that nothing more needs to be verified about them for the specific property; and (*ii*) formally prove that the remaining critical parts satisfy the targeted property. The key aspect of the vision is the *system-level* specification of the property of interest, and the incorporation of all critical code in an overall proof, including the kernel. A challenge will be to keep the security-critical parts or *trusted computing base* (TCB) as small and simple as possible to ensure that its verification remains tractable.

Our vision builds on, and is enabled by, the formal verification of the seL4 microkernel. Microkernel-based systems already componentise software into smaller, isolated, components for security, safety, or reliability. SeL4's verification will eventually enable provable isolation guarantees by providing correct kernel mechanisms for managing the hardware platform's memory protection mechanisms.

The remainder of the paper presents our first steps towards realising our vision for large, secure systems on seL4. We use a concrete case study of a *secure access controller* (SAC) as a representative example of a large complex system with a specific property requirement. Section 2 describes the SAC in more detail. Section 3 briefly overviews seL4, and presents a SAC design (and rationale) that is architected to minimise the TCB. Section 4 describes how to formally verify security prop-

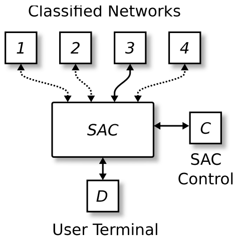

Figure 1: The SAC routes between a user's terminal and 1 of n classified networks.

erties on that architecture such that the properties still hold at the implementation level, and includes the formalisation of the information flow property targeted for the SAC and its proof using Isabelle/HOL. Finally, Section 5 looks at related work, while Section 6 concludes.

2 Case study overview

To illustrate some of the difficulties present in verifying large systems, we introduce a case study of a simple SAC device. In this scenario, a single user requires access to several independent networks of different security classifications. The user has a simple terminal connected to a network interface of the SAC. The SAC has additional network interfaces allowing it to be connected to each of the classified networks. The user only needs to access one network at a time, and selects the network through a web interface provided by the SAC on a control network interface. This setup is depicted in Figure 1.

The goal of the SAC is to route TCP/IP packets between the user's terminal and the currently selected network without allowing the information to be seen or manipulated by the other networks. The SAC must ensure that all data from one network is isolated from each of the other networks. While we assume that the user's terminal is trusted to not leak previously received information back to another network, we otherwise assume that all networks connected to the SAC are malicious and will collude.

Concrete applications of such a device can be found in the defence sector, where users frequently need to deal with data of several classifications, each of which is isolated on its own network. The traditional approach of having one terminal per classification level for each user, while clearly obeying the security requirements, is rather unwieldy.

While the requirements of the SAC are quite simple, it already presents several challenges to full system verification. In particular, the SAC requires code for (*i*) gigabit network card drivers; (*ii*) a secure web server; (*iii*) a TCP/IP stack for the web server; and (*iv*) IP routing code. Any one of these components would individually consist of tens of thousands of lines of non-trivial code that would give even the most seasoned verification engineer pause. Complicating matters further, each of the classified networks needs to both read and write data to the user's terminal at some point in time. Traditional data diodes or any security design that relies on statically partitioning resources would be incapable of providing the required functionality of the SAC. Despite these complications, our goal is to provide the required functionality while having a full system assurance that the data from the networks will remain isolated.

3 Designing for the Vision

For our case study, our property of interest is an access-control-based security property. Verifying such a property for the large body of code needed to implement the functionality required by the SAC is far beyond the abilities of current verification methods.

To overcome this we split the code of a large system into two classes: *trusted* code, implementing security-critical functionality, and *untrusted* code which we assume is malicious, avoiding the need to reason about its precise implementation. For such a split to be possible, we need some mechanism that allows such untrusted code to be securely isolated.

Our work uses the seL4 microkernel to provide such isolation. SeL4 is a small operating system kernel of the L4 family designed to be a secure, safe, and reliable foundation for a wide variety of application domains [11]. Its C implementation has been formally proved to match its functional specification [7], making it a key foundation of our goal for full system assurance. As a microkernel, it provides a minimal number of services to applications: abstractions for virtual address spaces, threads and inter-process communication (IPC).

SeL4 uses a capability-based access-control model. All memory, devices, and microkernel-provided services require an associated *capability* (access right) to utilise them [3]. The set of capabilities a component possesses determines what a component can directly access. SeL4 enforces this access control using the hardware's memory management unit (MMU). Additionally, seL4 allows device drivers to be isolated by using the I/O MMU functionality present on recent x86 processors. The I/O MMU allows the kernel to control what areas of physical memory each hardware device can access via *direct memory access* (DMA), preventing malicious hardware devices (or, more specifically, malicious software controlling such hardware devices) from bypassing seL4's access control mechanisms.

The access control mechanism of seL4 allows systems

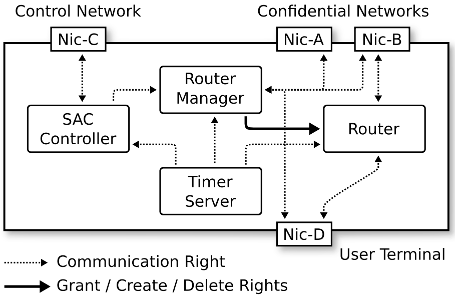

Figure 2: High-level component breakdown of the SAC design. The router manager is the only trusted component in the system, as no other component has simultaneous access to both NIC-A and NIC-B.

to be broken into smaller independent components, each with its own set of access rights. This split of components forms the system's high-level *security architecture*. The set of capabilities we provide to each component forms the system's *capability distribution* which precisely defines overt communication amongst components and hardware, and thus can be used as the basis of a security analysis of the system. Covert communication channels, such as timing channels, would have to be analysed by other means.

Components that do not possess any capabilities that may be used to violate the system's security policy need not be trusted, and can be implemented without requiring verification. Components that *do* have sufficient capabilities to violate the system security policy become part of the TCB (along with the seL4 kernel itself), and require verification. For example, in our SAC case study, any component that possesses a capability to a network card connected to a classified network while simultaneously having access to information from another classified network would need to be trusted, and hence verified.

In our experience, designing a secure system is an iterative process: (*i*) a high-level security architecture is proposed, coarsely breaking the system down into components; (*ii*) a capability distribution is determined by applying the principle of least privilege for the design [10]; and (*iii*) this capability distribution is analysed to determine which components have sufficient rights to violate the desired security policy (hence becoming part of the system's TCB). The resulting design may be further refined via re-iterating to reduce the size of the TCB, and thus ease verification effort.

We return to the case study to illustrate how this design process applies to the SAC. For simplicity of explanation, we assume that the SAC only needs to multiplex two classified networks, NIC-A and NIC-B. The user's terminal is connected to NIC-D, while the SAC is controlled through a web interface provided on NIC-C.

To avoid trusting (and thus verifying) large bodies of code such as network stacks, we architect the system with an untrusted *router* component. This component is given access to NIC-D and either NIC-A or NIC-B, and is responsible for routing between the two networks. The component has two additional parts: read-only access to its own initialisation code and additional read-write memory required by it at run-time.

A second trusted component, the *router manager*, possesses capabilities to all three of NIC-A, NIC-B and NIC-D. When the SAC needs to switch between networks, the router manager first deletes any running router component, clears the router's read-write memory, and sanitises the hardware registers and buffers of NIC-D (to prevent any residual information from inadvertently being stored in it). Such sanitisation requires a detailed knowledge of the the network card hardware (to ensure that all potential storage channels are cleared), but is expected to be significantly simpler than an implementation of a full driver for the card. The router manager will then recreate the router, grant it access to its read-only code and read-write memory, and grant it access to NIC-D and either NIC-A or NIC-B as required. This allows the router to switch between NIC-A and NIC-B without being capable of leaking data between the two.

A third untrusted component, the *SAC controller*, provides a web interface to the control network on NIC-C. The router manager is given a read-only communication channel to the SAC controller, which is used to instruct the router manager to restart the router with rights to the other classified network.

Finally, to avoid components sharing the system's timing hardware (thus creating a communications channel between them), a fourth untrusted *timer server* component is granted access to the system clock and provided with a write-only communication channel to each of the other components. It broadcasts a regular timer tick to the other components, allowing each to internally track time, required by modern network card drivers and TCP/IP implementations.

This design, shown in Figure 2, only requires the router manager to ever have access to both NIC-A and NIC-B simultaneously. While this means that the router manager component becomes part of the system's TCB, it allows us to leave all other components in the system untrusted, significantly easing the burden of verification.

Our implementation of this design uses GNU/Linux to implement the router and SAC controller components. The SAC controller's webserver is implemented using '`mini_httpd`', while the Linux kernel itself provides functionality for routing, the TCP/IP networking stack and drivers for the network cards. The Linux kernel alone consists of millions of lines of code, much of which would become part of the TCB if used directly. By utilising the access control features of seL4 and designing the system to isolate this functionality, we were able to reduce the run-time TCB of the SAC to just the router manager (approximately 1500 lines of code) and the seL4 kernel (approximately 7500 lines of code), just under 9000 lines in total.

4 Formal verification of security properties

While the previous section described informally how a secure system such as the SAC might be designed to reduce the size of its TCB, this alone does not provide any guarantees about our desired security property. This section describes a process that allows us to formally prove that the final system implementation obeys the property, and describes our progress on this vision by describing the first few steps of the proof on our SAC case study.

As in seL4's correctness proof, we focus on verification of the initialized C code, assuming the correctness of hardware, compiler, assembly and booter (the two latest being on-going work).

This verification approach is illustrated in Figure 3. Once a system has been broken into components with an initial capability distribution defined (labelled ① in the figure) and trusted components in the system have been identified (labelled ② in the figure), we must then:

1. Prove that this partition is sound. That is, we must prove that untrusted components are incapable of violating the targeted security property of the system. This is done by describing the behaviour of trusted components (labelled ③ in the figure), and modelling untrusted components as capable of carrying out any series of actions authorised by the set of capabilities they possess. If, under these assumptions, a proof of security succeeds, nothing further needs to be proven about the untrusted components.

2. Prove that the code of the run-time TCB (i.e., the trusted components and underlying kernel) correctly implements the security model used for the proof. This involves taking the simple model used to perform the proof in step 1, and then refining it (possibly via several increasingly more precise models), down to the final system's implementation.

The second step involves three tasks, most of them being on-going or future work. First, we need to prove that the kernel implementation refines its security model. Building on seL4's proof of correctness reduces this task to proving that the high-level specification the kernel implements refines the security model. This is on-going work. The second task is to prove that the trusted components' implementations refine their formal behavior. This has not been done for the SAC system but our experience from the kernel verification and the framework built for such refinement give us confidence that this task is feasible. The last task consists in proving that the initial capability distribution in the system implementation satisfies the abstract security architecture. We have defined a capability distribution language, called capDL [8], with a formal semantics that aims to be used to automatically and formally link a user-defined capability distribution description of the system to both an initial implementation state and an abstract security architecture.

The remainder of this section describes the first step of the two listed above in detail, illustrating them with our SAC case study.

4.1 Notation

We briefly introduce the notation used for the remainder of this paper. Our meta-language Isabelle/HOL conforms for the most part with normal mathematical notation.

The space of total functions is denoted by $\Rightarrow$. Type variables are written $'a$, $'b$, etc. The notation $t :: \tau$ means that HOL term t has HOL type τ. The option type **datatype** $'a\ \mathit{option} = \mathit{None} \mid \mathit{Some}\ 'a$ adjoins a new element None to a type $'a$. Function update is written $f(x := y)$ where $f :: {'a} \Rightarrow {'b}$, $x :: {'a}$ and $y :: {'b}$ and $f(x \mapsto y)$ stands for $f\ (x := \mathit{Some}\ y)$.

Isabelle supports tuples with named components. For instance, we write **record** $\mathit{point} = x :: \mathit{nat},\ y :: \mathit{nat}$ for the type point with two components of type nat. If p is a point, a possible value for p is notated $(\!|x{=}5{,}y{=}2|\!)$. The term $x\ p$ stands for the x-component of p. Updating p from a current value $(\!|x{=}5{,}y{=}2|\!)$, with the update notation $p(\!|x{:=}4|\!)$, gives $(\!|x{=}4{,}y{=}2|\!)$. Finally, the keyword **types** introduces a type abbreviation.

4.2 Underlying access control model

From a security point of view, the operations provided by the kernel can be reduced to seven possible operations: read, write, create, delete, remove, grant, revoke and four corresponding access rights: read (r), write (w), create (c) and grant (g). The seL4 kernel supports more operations, but purely from the security perspective, each of them can be reduced to a sequence of these seven. For instance an IPC receive can be reduced to a read, while an IPC send can be reduced to a write.

The first five operations allow a component to read or write from another component, to create a new component, to delete an existing component, or to remove an

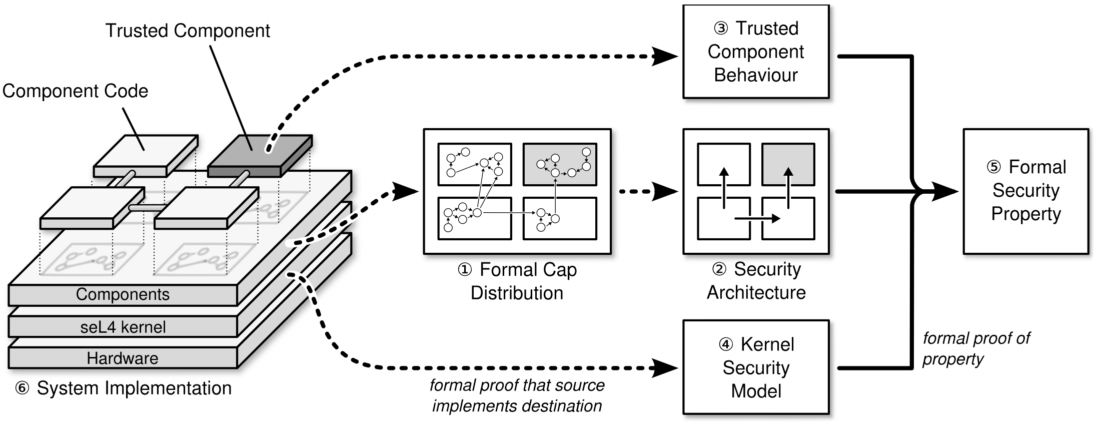

Figure 3: Full-system verification approach for seL4-based system

existing capability. All of these operations require the acting component to hold the correct capability to the target entity. The last two operations, grant and revoke, enable a component to delegate one of its capabilities to another component or to withdraw one or more capabilities from other components. Whereas the grant operation requires an explicit capability for authorisation as above, the revoke operation is authorised implicitly. Each component may revoke any capability it has created and all copies it has granted as long as it still holds the original capability. The kernel internally tracks this create/grant relationship in an internal book-keeping mechanism, and authorises revoke and delete operations accordingly.

In the following, we define the state space the application level entities will operate in, together with the transitions on this state space that the seL4 kernel allows.

4.2.1 State Space Model

Our model is largely inspired by seL4's security model developed in previous work [4, 5, 2]. In this previous work, all the kernel objects (active and inactive) are modelled as *entities*, and the state space only stores the capabilities each entity in the system has access to. In other words, it abstracts away from all application-local or kernel-internal storage, and instead concentrates on how capabilities—and therefore access to information—are distributed throughout the system. The only extension made here is regarding storage: for certain security properties we may need to track additional state.

For instance, in our case study, the property we are interested in for the SAC is the absence of explicit information flow, i.e., confidential data being explicitly read by an external entity. For simplicity, we only aim to prove that there is no information flow from NIC-A to NIC-B (the property being symmetric). The *confidential data* is therefore the data coming from NIC-A and the *external entity* that should not obtain any information from this confidential data is NIC-B. The approach taken is to tag the data coming from NIC-A as confidential. This means that we give any entity with storage (memory and network cards) a flag denoting whether it could possibly contain data from NIC-A. Entities that have this flag set are called *contaminated*. In their initial state, no entity, other than NIC-A, is contaminated. Each time an entity reads from a contaminated entity, it too becomes contaminated. Likewise, if a contaminated entity performs a write operation to another entity, the target becomes contaminated. The goal then is to prove that the whole SAC can never reach a state where NIC-B is contaminated.

Entities are therefore represented by the set of capabilities they hold and their "contamination status":

record *entity* =
  caps :: *cap set*
  contam :: *bool*

Each capability contains a reference to an entity it grants rights to, and the set of access rights it provides:

datatype *right* = *Read* | *Write* | *Grant* | *Create*
record *cap* =
  entity :: *entity-id*
  rights :: *right set*

Both the capability set and each entity's contamination state can dynamically change. The state of the system at a given point is a function from entity identifiers to entities. We model the fact that not all entity identifiers are mapped to entities by using the option type:

datatype *entity-id* =
  SacController | *NicA* | *NicB* | *NicC* | *NicD*
  | *RouterManager* | *Router* | *RouterMem* | *RouterCode*
  | *Timer* | *TimerChip* | *UnknownEntity nat*.

types *state* = *entity-id* ⇒ *entity option*

Note that the entity's contamination status can be generalised to other kinds of storage information required by a label-based security property.

4.2.2 System operations

The possible basic transitions on this state space are described by the kernel operations available to components.

We do not model the revoke operation in its general case here, but instead represent it by the specific sequence of remove operations that eventually take place. The operations' formalization is the following:

datatype *sys-op* =
  SysRead cap | *SysWrite cap bool*
  | *SysCreate cap* | *SysGrant cap cap*
  | *SysDelete cap* | *SysRemoveSet cap* (*cap set*)

All of the operations take a capability pointing to the targeted entity. In the case of *SysRead c* for instance, the entity performing the operation is reading from the entity referred to by the capability *c*. The operation will only be allowed by the kernel if the capability *c* is held by the entity performing the operation and includes at least the read right. *SysGrant* also takes the capability to be granted and *SysWrite* takes a boolean flag which is true if the write operation is a *flush* operation, removing an entity's contamination flag. Such an operation is required to model the router manager's sanitisation of NIC-D when the network is being switched. The final operation *SysRemoveSet* removes a set of capabilities.

The authorisation check for all the system operations is summarised in the function *legal s e sysop* defining the conditions for entity *e* to perform operation *sysop* in state *s*. For instance:

$\mathit{legal}\ s\ e\ (\mathit{SysRead}\ c) =$
$\quad (\mathit{is\text{-}entity}\ s\ e \wedge \mathit{is\text{-}entity}\ s\ (\mathit{entity}\ c) \wedge$
$\quad c \in \mathit{entity\text{-}caps\text{-}in\text{-}state}\ s\ e \wedge \mathit{Read} \in \mathit{rights}\ c)$

where *is-entity* ensures that entity *e* is defined in *s* and *entity-caps-in-state* retrieves the capabilities held by *e*.

4.2.3 State transitions

We now look at how the state changes for each operation. This is modelled by the function *step s e sysop* defining the resulting state after the entity *e* has performed operation *sysop* on state *s*. For instance, we model both reads and writes as a *write-operation* (with the direction of the write switched for reads):

$\mathit{step}\ s\ e\ (\mathit{SysWrite}\ c\ b) = \mathit{write\text{-}operation}\ e\ (\mathit{entity}\ c)\ b\ s$
$\mathit{step}\ s\ e\ (\mathit{SysRead}\ c) = \mathit{write\text{-}operation}\ (\mathit{entity}\ c)\ e\ \mathit{False}\ s$

where *write-operation* is defined as follows:

$\mathit{write\text{-}operation}\ \mathit{source}\ \mathit{target}\ \mathit{is\text{-}flush}\ \mathit{ss} \equiv$
$\quad (\mathit{case}\ \mathit{ss}\ \mathit{target}\ \mathit{of}$
$\quad\ \ \mathit{Some}\ \mathit{target\text{-}entity} \Rightarrow$
$\quad\quad \mathit{ss}(\mathit{target} \mapsto \mathit{target\text{-}entity}(\!|\ \mathit{contam} :=$
$\quad\quad\ \ ((\mathit{is\text{-}contam}\ \mathit{ss}\ \mathit{target} \vee \mathit{is\text{-}contam}\ \mathit{ss}\ \mathit{source})$
$\quad\quad\quad \wedge \neg \mathit{is\text{-}flush})|\!))$
$\quad\ \ |\ \text{-} \Rightarrow \mathit{ss})$

The other operations are defined similarly and are used to define *step*. A *legal-step* is defined as a step of the system that only takes place if it is legal. If the operation can not be performed (because the thread attempting the operation doensn't have an appropriate capability, for instance), the operation silently fails and the system state remains unchanged:

$\mathit{legal\text{-}step}\ s\ \mathit{e\text{-}id}\ \mathit{sysop} \equiv$
$\quad \mathit{if}\ \mathit{legal}\ s\ \mathit{e\text{-}id}\ \mathit{sysop}\ \mathit{then}\ (\mathit{step}\ s\ \mathit{e\text{-}id}\ \mathit{sysop})\ \mathit{else}\ s$

This model, for simplicity, does not allow threads to determine if an operation failed; trusted threads need to ensure that they have the correct resources before attempting any security-critical operation.

4.3 Component-level model

So far we have described the states and transitions of the underlying kernel which the components will run on. This model is used to describe the components' behavior as sequences of instructions, where each instruction is a *step* modifying the global state of the system. As explained earlier, we model only the trusted components' behavior. No restriction at all is placed on the untrusted components, and they will be correctly implemented by any concrete program code. Their behaviour is only constrained by the authority they are given via capabilities. In our case study this means that the router instance, when it has received the capabilities to its network cards, will be able to attempt any behaviour, but the kernel will only allow access to the two network cards (NIC-A and NIC-D, say) it possesses capabilities to at this point. If we can show that the system is secure with this unconstrained behaviour, it will also be secure with any specific implementation of the router components.

While the specification of untrusted components is simple, the specification of trusted components requires more care. We rely on specific behaviour of the trusted component for the security of the overall system. In our case study, we rely on the router manager to execute specific operations in a specific order, such as creating the router instance, granting capabilities, revoking capabilities, flushing the network cards, etc.

We model the program of such trusted entities as a list of instructions, each of which either performs a kernel operation (*SysOp*) or changes the program counter of the entity (*Jump*). To avoid needing to reason about implementation details of trusted entities, flow control (such as ‘`if`’ and ‘`case`’ statements) is modelled by non-deterministic choice, which itself is modelled by having the *Jump* instruction accept a list of targets. Untrusted entities may perform any operation they wish, so are modelled with a program consisting of an *AnyOp* instruction, representing any legal kernel operation:

```
datatype instruction =
    SysOp sys-op
  | Jump nat list
  | AnyOp
types program = entity-id ⇒ instruction list
```

To model the behavior of the whole system, we need to represent the fact that entities run concurrently. We model this behaviour by considering all possible interleavings of instructions between entities. For this, we keep track of a program counter for each component in an additional program counter state:

```
record sys-state =
  sys-entity-st :: state
  sys-pc-st :: entity-id ⇒ nat
```

One execution step of the whole system consists of non-deterministically choosing any existing active entity and running its current instruction (specified by its program counter). This models the seL4 kernel scheduler. If the current instruction is *AnyOp*, then we pick any arbitrary operation that is legal in the current state for this entity and execute it. We thus get a safe over-approximation of all possible execution traces of the system.

To model a single step of a particular entity *e-id*, we look at the instruction at that entity's program counter. If it is a *SysOp* or *AnyOp* operation, it is executed using *legal-step* producing a new state of the system. If the instruction is a *Jump* operation, the model non-deterministically updates the current entity's program counter to one of the values in the list *loffset*. In both cases, the model requires that the entity performing the instruction exists and that the entity's program counter is within the bounds of its program:

```
inductive
  entity-operation :: entity-id ⇒ sys-state ⇒ sys-state ⇒ bool
where
  entitySysOp:
  ⟦ ss = ⦇ sys-entity-st = s, sys-pc-st = pc ⦈;
    is-entity s e-id; pc e-id = e-pc;
    sys-program e-id = e-prog; e-pc < length e-prog;
    e-prog ! e-pc = SysOp oper ∨ e-prog ! e-pc = AnyOp;
    s' = legal-step s e-id oper;
    new-pc = (e-pc + 1) mod (length e-prog);
    ss' = ⦇sys-entity-st=s',sys-pc-st=pc(e-id:=new-pc)⦈⟧
    ⟹ entity-operation e-id ss ss'
  | entityJump:
  ⟦ ... (∗ as above ∗)
    e-prog ! e-pc = Jump loffset;
    new-pc ∈ set loffset;
    ss' = ⦇sys-entity-st=s,sys-pc-st=pc(e-id:=new-pc)⦈⟧
    ⟹ entity-operation e-id ss ss'
```

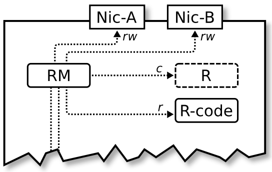

Figure 4: SAC initial state (partial)

A single execution step of the whole system is then modelled by the relation $ss \rightarrow ss'$ which is true if ss' is a possible resulting system state after executing the current instruction of any existing active entity in the system state *ss*. An execution $ss \rightarrow^* ss'$ is then defined as a sequence of execution steps.

We have now defined execution and implicitly with the relation above all possible execution traces of the system.

Instantiation to a given system. This formalisation of seL4-based systems' behavior can be instantiated to a specific componentised system like the SAC. This is done by defining the initial capability distribution for this system and the program of each of its trusted components. For instance, the initial state for the SAC system (partially illustrated in Figure 4) is modelled as:

```
SAC-startup≡ ⦇sys-entity-st=SAC-init-state, sys-pc-st=λx. 0⦈
```

where *SAC-init-state* defines the initial capability set for each component, together with their contamination status (of which all components other than NIC-A are initially uncontaminated). For instance, the router manager's initial state looks like:

```
RM0 ≡ ⦇ caps = { cap-RW-to-NIC-A, ...}, contam = False ⦈
```

where we take the convention that the name of each cap is of the form: *cap-<rights>-to-<target-entity>*, as in:

```
cap-RW-to-NIC-A ≡ ⦇ entity=NicA, rights = {Read, Write} ⦈
```

Each trusted component's behaviour is modelled as a sequence of instructions. For instance, the router manager in our case study will be formalized as follows.

```
RM-prg ≡
  [(∗ 00: Wait for command, delete router manager. ∗)
  SysOp (SysRead cap-R-to-SAC-C),
  SysOp (SysRemoveAll cap-C-to-R),
  SysOp (SysDelete cap-C-to-R),
  SysOp (SysWriteZero cap-RW-to-NIC-D),
  ...
  Jump [0, 10, 19],
  (* 10: Setup router between NIC−A and NIC−D. *)
  SysOp (SysCreate cap-C-to-R),
  SysOp (SysNormalWrite cap-RWGC-to-R),
  SysOp (SysGrant cap-RWGC-to-R cap-RW-to-NIC-A),
  SysOp (SysGrant cap-RWGC-to-R cap-RW-to-NIC-D),
  SysOp (SysGrant cap-RWGC-to-R cap-R-to-R-code),
  ... ]
```

The *sys-program* function that associates a program to each component (used *entity-operation*) is defined as:

```
sys-program eid ≡
  if (eid = RouterManager) then RM-prg
  else if (eid ∈ untrusted-entities) then [AnyOp] else []
```

where *untrusted-entities* for the SAC consist of *SacController*, *Router*, *Timer* and where the inactive entities (such as the network cards) are associated with empty programs.

4.4 Security property proof

With the model described, we can now formally state the security property we are targeting for our SAC case study. The property we are interested in is the absence of explicit information flow. As explained earlier, we model the fact that NIC-B cannot read information from NIC-A in a given state as NIC-B not being contaminated. In particular, we state that in any state that the SAC can reach starting from its initial state, NIC-B is not contaminated with data from NIC-A:

```
lemma sacSecurity: ⟦ SAC-startup →* ss′ ⟧ ⟹
  ¬ is-contaminated (sac-entity-st ss′) NicB
```

The proof relies on showing an invariant always holds on the state of the SAC. The invariant insists that: (*i*) Only NIC-A, the router (and associated components) and NIC-D ever become contaminated; (*ii*) The capabilities held by each component is limited to a small, secure set; (*iii*) The router doesn't have capabilities to both NIC-A and NIC-B at the same time; (*iv*) The router doesn't have a capability to NIC-B while any component it can access is contaminated; (*v*) The capabilities held by the router manager at every point of time is sufficient to allow it perform its job of deleting the router and sanitising NIC-D; (*vi*) All entities other than the router always exist; and finally (*vii*) That certain conditions about the state of the router hold when the router manager's program counter is at particular values.

The last invariant is the most intricate, and is required to show that the system remains in a well-known state while the router manager is mid-way through deleting or creating the router instance. For instance, when the router manager's program counter points to an instruction granting the router access to NIC-B, we must know that the router is in an uncontaminated state, which can only be established because the router manager earlier deleted the router, and hasn't provided it with any caps to NIC-A since.

The final security property follows directly from the invariant, which states that NIC-B will always be uncontaminated.

5 Related Work

The idea of using system architectures to ensure security by construction, relying on basic kernel mechanisms to separate trusted from untrusted code is widely explored in the MILS (multiple independent levels of security and safety) space [1]. In the context of formal analyses of capability-based software, Spiessens [12] developed the formal language *Scoll* to model the behaviour of trusted components, together with a model-checker for that language to check capability-based software.

Murray [9] builds on Spiessens' concepts, but uses a CSP, for which model-checking tools already exist. Its main contribution is to extend the kind of properties that can be expressed to also include noninterference style information flow properties (under the assumption that the capability system that the software is running on does not expose covert channels between unconnected objects) and liveness properties under fairness assumptions.

Both Spiessens' and Murray's work explicitly prove that it is safe to take multiple entities and model them as a single entity that possesses the union of their capabilities and exhibits the union of their behaviours. This idea is also part of our vision, with the addition of a capability abstraction (the capability distribution in Figure 3 to reason about typed capabilities upon kernel objects, whereas at the security architecture level, the simple model of read, write, create, grant capabilities between components is used). However, the proof that it is safe to aggregate entities in this way is part of our future work.

To the best of our knowledge, our work is the first to use interactive theorem proving rather than model-checking to verify capability based systems by modelling trusted components' behaviour. We also modelled the SAC using the SPIN model checker [6] for comparison. Although the proof effort was reduced from around six weeks (by an inexperienced Isabelle/HOL user) to less than one day, our model quickly reached a size that was beyond the abilities of SPIN to verify in a reasonable amount of time and memory. In particular, we could only verify our final system design by making simplifying assumptions in the SPIN model. The other main benefit of using a theorem prover is being provided with a framework to prove the refinement between the security architecture and trusted components' behaviour, and the system implementation. Assuring label-based security properties about the actual implementation of real-world system is the real added value of the framework we propose. Investigating how interactive theorem proving and model checking can be combined in a way that gives the flexibility of the former with the ease-of-use of the latter is part of our future work.

6 Conclusion

In this paper we have presented our vision of how large software systems consisting of millions of lines of code can still have formal guarantees about certain targeted properties. This is achieved by building upon the access control guarantees provided by the verified seL4 microkernel and using it to isolate components such that their implementation need not be reasoned about.

We have demonstrated in our SAC case study how careful design and componentisation of a large system can be used to reduce the run-time TCB from millions of lines of code to just under 9000. Additionally, we have modelled the design of the SAC and shown that the modelled system fulfills its security goal of isolating data between different networks.

What still remains is connecting the model used to prove security of the system with the actual implementation. In particular, we must still show that (*i*) the C implementation of trusted components in the SAC refine the behaviour modelled in the security proof; and (*ii*) that the kernel operations in the security proof correctly model the actual behaviour of the seL4 kernel. The verification success of the seL4 kernel, with its C code shown to implement its functional specification, gives us confidence that both of these tasks are feasible. Carrying out this verification effort forms part of our ongoing work.

Acknowledgements The authors would like to thank Gerwin Klein, Toby Murray and Simon Winwood for their advice, feedback and contributions to this paper, and Xin Gao for his contributions to the proof.

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.

This material is in part based on research sponsored by the Air Force Research Laboratory, under agreement number FA2386-09-1-4160. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

References

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

[2] A. Boyton. A verified shared capability model. In G. Klein, R. Huuck, and B. Schlich, editors, *4th SSV*, volume 254 of *ENTCS*, pages 25–44, Aachen, Germany, Oct 2009. Elsevier.

[3] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *CACM*, 9:143–155, 1966.

[4] D. Elkaduwe, G. Klein, and K. Elphinstone. Verified protection model of the seL4 microkernel. Technical Report NRL-1474, NICTA, Oct 2007. Available from http://ertos.nicta.com.au/publications/papers/Elkaduwe_GE_07.pdf.

[5] D. Elkaduwe, G. Klein, and K. Elphinstone. Verified protection model of the seL4 microkernel. In J. Woodcock and N. Shankar, editors, *VSTTE 2008*, volume 5295 of *LNCS*, pages 99–114, Toronto, Canada, Oct 2008. Springer.

[6] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[7] 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. seL4: Formal verification of an OS kernel. In *22nd SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.

[8] I. Kuz, G. Klein, C. Lewis, and A. Walker. capDL: A language for describing capability-based systems. In *1st APSys*, New Delhi, India, Aug 2010. To appear.

[9] T. Murray. *Analysing the Security Properties of Object-Capability Patterns*. PhD thesis, University of Oxford, 2010.

[10] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), 1975.

[11] seL4 Website. http://ertos.nicta.com.au/research/sel4/, Jun 2010.

[12] A. Spiessens. *Patterns of Safe Collaboration*. PhD thesis, Université catholique de Louvain, Louvain-la-Neuve, Belgium, February 2007.

From a Verified Kernel Towards Verified Systems

Gerwin Klein

[1] NICTA, Australia

[2] School of Computer Science and Engineering, UNSW, Sydney, Australia

gerwin.klein@nicta.com.au

Abstract. The L4.verified project has produced a formal, machine-checked Isabelle/HOL proof that the C code of the seL4 OS microkernel correctly implements its abstract implementation. This paper briefly summarises the proof, its main implications and assumptions, reports on the experience in conducting such a large-scale verification, and finally lays out a vision how this formally verified kernel may be used for gaining formal, code-level assurance about safety and security properties of systems on the order of a million lines of code.

1 L4.verified

In previous work [13], we reported on the result of the L4.verified project: a machine-checked, formal verification of the seL4 operating system microkernel from a high-level model in Higher-Order logic down to low-level C code.

To the best of our knowledge, this is the first complete code-level proof of any general-purpose OS kernel, and in particular the first machine-checked such proof of full functional correctness.

Early pioneering attempts at formal OS verification like UCLA Secure Unix [20] or PSOS [9] did not proceed substantially over the specification phase. In the late 1980s, Bevier's KIT [2] is the first code-level proof of an OS kernel, albeit only a very simple one. There have been a number of formal verifications of either functional correctness, temporal, or information flow properties of OS kernels, recently for instance the Common Criteria EAL6+ certified INTEGRITY kernel [18]. None of these, however, truly formally verified the code-level implementation of the kernel. Instead, what is verified is usually a formal model of the code, which can range from very precise as in the INTEGRITY example to design-level or more abstract models. Correspondence between C code as seen by the compiler and the formal model is established by other means. In the L4.verified project, this critical missing step is for the first time formal and machine-checked.

Contemporary OS verification projects include Verisoft, Verisoft XT, and Verve. The Verisoft project has not yet fully completed all parts of its OS kernel proof, but it has conclusively demonstrated that formal verification of OS code can be driven down to verified hardware — similarly to the verified CLI stack [3] from the 1980s, but going up to a verified C0 compiler with support for inline assembly and up to substantial scale. The Verisoft XT project [7] has demonstrated that the technology exists to deal with concurrent C at a scale of tens of thousands lines of code. The Verve kernel [22] shows that type and memory safety properties can be established on the assembly level via type systems and therefore with much lower cost. Verve contains a formally verified runtime system, in particular a garbage collector that the type system relies on. Even though it only shows type safety, not functional correctness, the smaller cost of verification makes the approach attractive for larger code bases if full functional correctness is not required or too expensive to obtain.

The formal proof for the seL4 kernel establishes a classical functional correctness result: all possible behaviours of the C implementation are already contained in the behaviours of its abstract specification. In the L4.verified project, this proof was conducted in two stages in the interactive theorem prover Isabelle/HOL [17]. The first stage is comparable to other detailed model-level kernel verifications. It connects an abstract, operational specification with an executable design specification of the kernel. This design specification is low-level enough to clearly see a direct one-to-one correspondence to C code for the large majority of the code. The second step in the proof was to show that the C code implements this low-level design. The result is one concise overall theorem in Isabelle/HOL stating that the behaviour of the C code as specified by its operational semantics is contained in the behaviours of the specification.

Like any proof, this verification has assumptions. For the correctness of a running seL4 system on real hardware we need to assume correctness of the C compiler and linker, assembly code, hardware, correct use of low-level TLB and cache-flushing instructions, and correct boot code. The verification target was the ARM11 uniprocessor version of seL4. There also exists an (unverified) x86 port of seL4 with optional multi-processor and IOMMU support.

The key benefit of a functional correctness proof is that proofs about the C implementation of the kernel can now be reduced to proofs about the specification if the property under investigation is preserved by refinement. Additionally, our proof has a number of implications, some of them desirable direct security properties. If the assumptions of the verification hold, we have mathematical proof that, among other properties, the seL4 kernel is free of buffer overflows, NULL pointer dereferences, memory leaks, and undefined execution. There are other properties that are not implied, for instance general security without further definition of what security is or information flow guaranties that would provide strict secrecy of protected data. A more in-depth description of high-level implications and limitations has appeared elsewhere [12,11].

2 What have we learned?

To be able to successfully complete this verification, we have contributed to the state of the art in theorem proving and programming languages on a number of occasions, including tool development [16], memory models [19], and scalable refinement frameworks [6,21]. These are published and do not need to be repeated in detail here. Other interesting aspects of the project concern lessons that are harder to measure such as proof engineering, teaching theorem proving to new team members, close collaboration between the kernel and verification teams, and a prototyping methodology for kernel development.

On a higher level, the main unique aspects of this project were its scale and level of detail in the proof. Neither would have been achievable without a mechanical proof assistant. The proof, about 200,000 lines of Isabelle script, was too large for any one person in the team to fully keep in their head, and much too large and technically involved to manually check and have any degree of confidence in the result. Software verifications like this are only possible with the help of tools.

The cost of the verification was around 25 person years counting all parts of the project, including exploratory work and models that were later not used in the verification. About twelve of these person years pertain to the kernel verification itself. Most of the rest was spent on developing frameworks, tools, proof libraries, and the C verification framework, including a precise memory model [19] and a C to Isabelle/HOL parser [21].

This means, we have demonstrated that proving functional correctness of low-level C code is possible and feasible at a scale of about 10,000 lines of code, but the cost is substantial. Clearly, we have to conclude that currently this approach does not lend itself to casual software development.

The story is different for high-assurance systems. It is currently very expensive to build truly trustworthy systems and to provide substantial assurance that they will indeed behave as expected. It is hard to get useful numbers for such comparisons, but one data point that is close enough, and where some experience and cost estimates are available, are Common Criteria (CC) security evaluations. CC on high evaluation levels prescribe the use of formal specifications and proofs down to the design level. Correspondence of models to code is established by testing and inspection.

L4.verified spent about \$700 per line of code (loc) for the verification if we take the whole 25 person years, and less than \$350/loc if we take the 12 actually spent on the kernel. We estimate that, with the experience gained and with the tools and libraries available now, the cost could be further reduced to 10, maybe 8 person years for a similar code base verified by the same team, i.e. about \$230/loc. Even assuming \$350/loc, the verification compares favourably with the quoted cost for CC EAL6 evaluation at \$1000/loc [10]. EAL7 (the highest CC level) which arguably still provides less assurance than formal code-level proof, can safely be assumed to be more costly still. The comparison is not entirely fair, since the Common Criteria mostly address security properties and not functional correctness, and because the verification aspect is only one of the aspects of the certification process. On the other hand one can argue that general functional correctness is at least as hard to prove as a specific security property and that while verification is not the only aspect, it is the most expensive one. We believe that formal, code-level verification is cost attractive for the vendor as well as for the certification authority, while increasing assurance at the same time.

For the certification authority, risk is reduced. Since the proof is machine-checked, only the high-level specification and its properties as well as the bottom-level model need to be scrutinised manually and with care to trust the system. Validating the high-level properties is the same as in the current evaluation scheme. The bottom-level model, however, is different. In the current scheme, the bottom level model is different for each certification and needs to be connected to code by careful validation, testing and inspection which is expensive to conduct and hard to check. In our case, the model does not depend on the certification artefact: it is just the semantics of our subset of C. Once validated, this could be re-used over many certifications and amortised to gain even higher assurance than what would otherwise be cost effective.

Our result of feasible but high-cost verification at about 10,000 loc does not mean that formal verification could not scale further. In fact, microkernels such as seL4 typically lack two properties that make formal verification scale better: modularity and strong internal abstractions. We would expect application-level code and even user-level OS code to be much better targets for scalable, compositional verification techniques.

However, even with nicely structured code, it appears infeasible at this stage to formally verify the functional correctness of systems with millions of lines of code. The field is making progress in scaling automated techniques for reasonably simple properties to such systems, but complex safety or security properties or properties that critically rely on functional correctness of at least parts of the system still appear without our reach.

3 A Secure System with Large Untrusted Components

This section presents a vision of how assurance even of complex safety properties could nevertheless be feasibly be achieved within (or close to) the current state of the art in code-level formal proof.

The key idea is the original microkernel idea that is also explored in the MILS (multiple independent levels of security and safety) space [4]: using system architectures that ensure security by construction, relying on basic kernel mechanisms to separate trusted from untrusted code. Security in these systems is not an additional feature or requirement, but fundamentally determines the core architecture of how the system is laid out, designed, and implemented.

This application space was one of the targets in the design of the seL4 kernel. Exploiting the verified properties of seL4, we should be able to architect systems such that the trusted computing base for the desired property is small and amenable to formal verification, and that the untrusted code base of the system provably cannot affect overall security.

The basic process for building a system in this vision could be summarised as follows:

1. Architect the system on a high level such that the trusted computing base is as small as possible for the security property of interest.

2. Map the architecture to a low-level design that preserves the security property and that is directly implementable on the underlying kernel.
3. Formalise the system, preferably on the architecture level.
4. Analyse, preferably formally prove, that it enforces the security property. This analysis formally identifies the trusted computing base.
5. Implement the system, with focus for high assurance on the trusted components.
6. Prove that the behaviour of the trusted components assumed in the security analysis is the behaviour that was implemented.

The key property of the underlying kernel that can make the security analysis feasible is the ability to reduce the overall security of the system to the security mechanisms of the kernel and the behaviour of the trusted components only. Untrusted components will be assumed to do anything in their power to subvert the system. They are constrained only by the kernel and they can be as big and complex as they need to be. Components that need further constraints on their behaviour in the security analysis need to be trusted to follow these constraints. They form the trusted components of the system. Ideally these components are small, simple, and few.

In the following subsections I demonstrate how such an analysis works on an example system, briefly summarise initial progress we have made in modelling, designing, formally analysing, and implementing the system, and summarise the steps that are left to gain high assurance of overall system security. A more detailed account is available elsewhere [1].

The case study system is a secure access controller (SAC) with the sole purpose of connecting one front-end terminal to either of two back-end networks one at a time. The back-end networks A and B are assumed to be of different classification levels (e.g. top secret and secret), potentially hostile and collaborating. The property the SAC should enforce is that no information may flow through it between A and B.

3.1 Architecture

Figure 1 shows the high-level architecture of the system. The boxes stand for software components, the arrows for memory or communication channel access. The main components of the SAC are the SAC Controller (SAC-C), the Router (R), and the Router Manager (RM). The Router Manager is the only trusted user-level component in the system. The system is implemented on top of seL4 and started up by a user-level booter component. The SAC Controller is an embedded Linux instance with a web-server interface to the front-end control network where a user may request to be connected to network A or B. After authenticating and interpreting such requests, the SAC Controller passes them on as simple messages to the Router Manager. The Router Manager receives such switching messages. If, for example, the SAC is currently connected to A, there will be a Router instance running with access to only the front-end data network card and the network card for A. Router instances are again embedded Linuxes

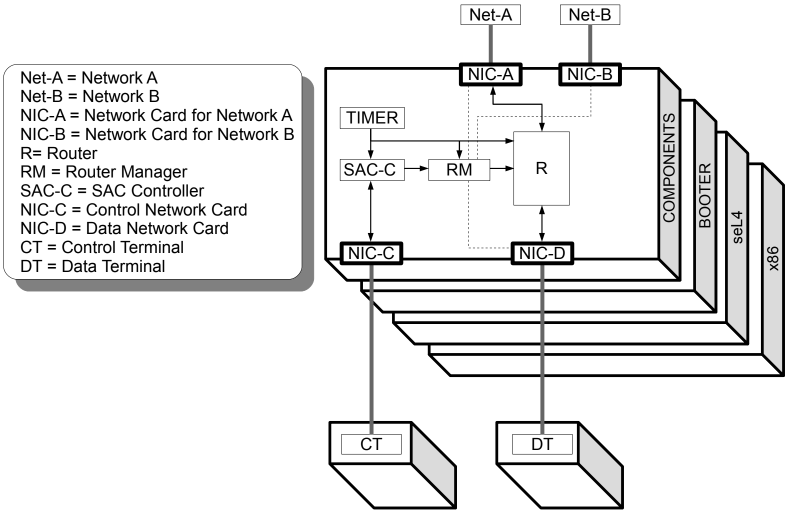

Fig. 1. SAC Architecture

with a suitable implementation of TCP/IP, routing etc. If the user requests a switch to network B, the Router Manager will tear down the current A-connected Linux instance, flush all network cards, create a new Router Linux and give it access to network B and the front end only.

The claim is that this architecture enforces the information flow property. Each Router instance is only ever connected to one back-end network and all storage it may have had access to is wiped when switching. The Linux instances are large, untrusted components in the order of a million lines of code each. The trusted Router Manager is small, about 2,000 lines of C.

For this architecture to work, there is an important non-functional requirement on the Linux instances: we must be able to tear down and boot Linux in acceptable time (less than 1-2 seconds). The requirement is not security-critical, so it does not need to be part of the analysis, but it determines if the system is practical. Our implementation achieves this.

So far, we have found an architecture of the system that we think enforces the security property. The next sections explore design/implementation and analysis.

3.2 Design and implementation

The main task of the low-level design is to take the high-level architecture and map it to seL4 kernel concepts. The seL4 kernel supports a number of objects for threads, virtual memory, communication endpoints, etc. Sets of these map to components in the architecture. Access to these objects is controlled by capabilities: pointers with associated access rights. For a thread to invoke any operation on an object, it must first present a valid capability with sufficient rights to that object.

Figure 2 shows a simplified diagram of the SAC low-level design as it is implemented on seL4. The boxes in the picture stand for seL4 kernel objects, the arrows for seL4 capabilities. The main message of this diagram is that it is significantly more complex than the architecture-level picture we started out with. For the system to run on an x86 system with IOMMU (which is necessary to achieve untrusted device access), a large number of details have to be taken care of. Access to hardware resources has to be carefully divided, large software components will be implemented by sets of seL4 kernel objects with further internal access control structure, communications channels and shared access need to be mapped to seL4 capabilities, and so forth.

The traditional way to implement a picture such as the one in Figure 2 is by writing C code that contains the right sequence of seL4 kernel calls to create the required objects, to configure them with the right initial parameters, and to connect them with the right seL4 capabilities with the correct access rights. The resulting code is tedious to write, full of specific constants, and not easy to get right. Yet, this code is crucial: it provides the known-good initial capability state of the system that the security analysis is later reduced to.

To simplify and aid this task, we have developed the small formal domain-specific language capDL [15] (capability distribution language) that can be used to concisely describe capability and kernel object distributions such as Figure 2. A binary representation of this description is the input for a user-level library in the initial root task of the system and can be used to fully automatically set up the initial set of objects and capabilities. Since capDL has a formal semantics in Isabelle/HOL, the same description can be used as the basis of the security analysis. It can also be used to debug, inspect and visualise the capability state of a running system.

For further assurance, we plan to formally verify the user-level library that translates the static capDL description into a sequence of seL4 system calls. Its main correctness theorem will be that after the sequence of calls has executed, the global capability distribution is the one specified in the original description. This will result in a system with a known, fully controlled capability distribution, formally verified at the C code level.

For system architectures that do not rely on known behaviour of trusted components, such as a classic, static separation kernel setup or guest OS virtualisation with complete separation, this will already provide a very strong security argument.

The tool above will automatically instantiate the low-level structure and access-control design into implementation-level C code. What is missing is providing the behaviour of each of the components in the system. Currently, components are implemented in C, and capDL is rich enough to provide a mapping between threads and the respective code segments that implement their behaviour. If the behaviour of any of these components needs to be trusted, this code needs to be verified — either formally, or otherwise to the required level of assurance.

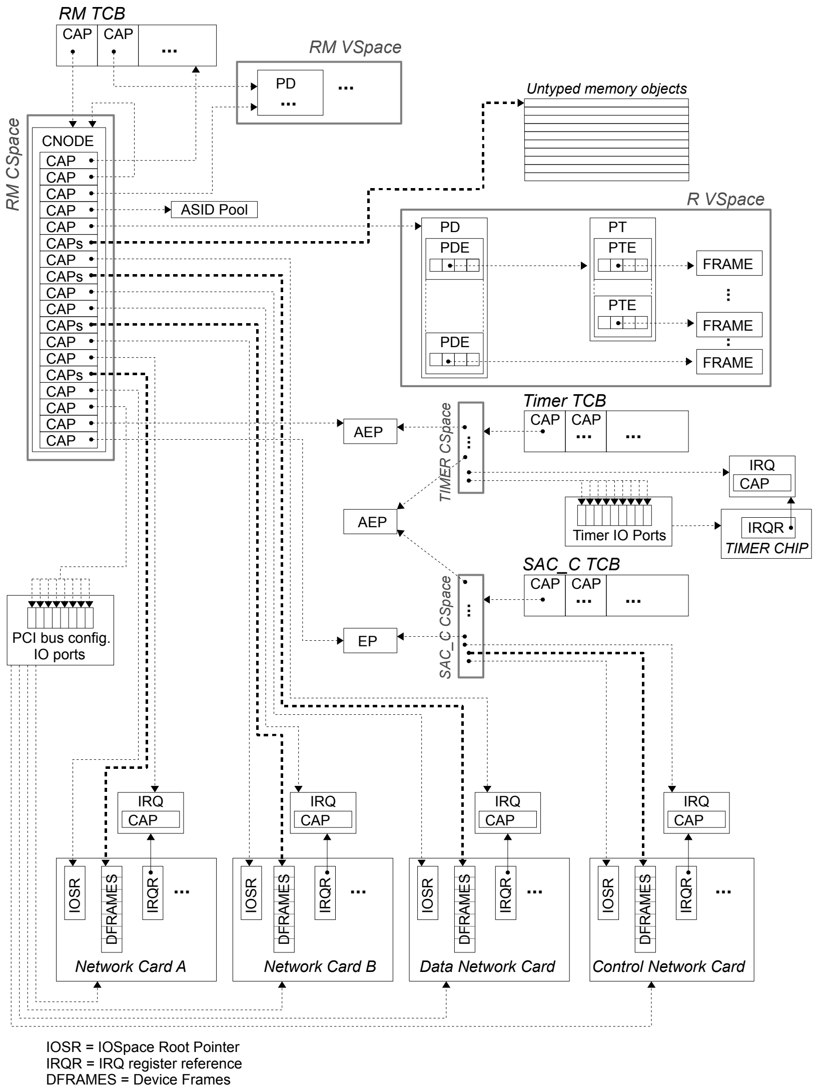

Fig. 2. Low-Level Design

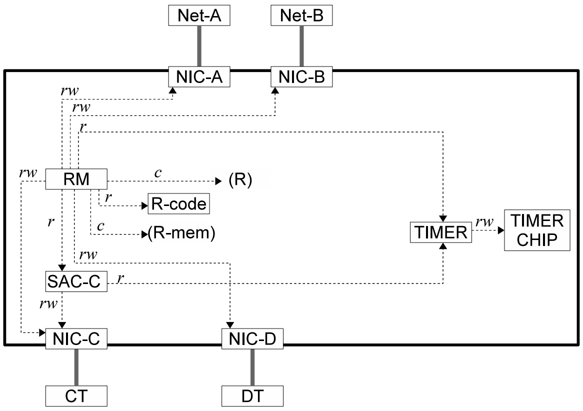

Fig. 3. SAC Abstraction

There is no reason component behaviour has to be described in C — higher-level languages such as Java or Haskell are being ported to seL4 and may well be better suited for providing assurance.

4 Security Analysis

Next to the conceptual security architecture of the SAC, we have at this stage of the exposition a low-level design mapping the architecture to the underlying platform (seL4), and an implementation in C. The implementation is running and the system seems to perform as expected. This section now explores how we can gain confidence that the SAC enforces its security property.

The capDL specification corresponding to Figure 2 is too detailed for this analysis. Instead, we would like to conduct the analysis on a more abstract level, closer to the architecture picture that we initially used to describe the SAC.

In previous work, we have investigated different high-level access control models of seL4 that abstract from the specifics of the kernel and reduce the system state to a graph where kernel objects are the nodes and capabilities are the edges, labelled with access rights [8,5]. We can draw a simple formal relationship between capDL specifications and such models, abstracting from seL4 capabilities into general access rights. We can further abstract by grouping multiple kernel objects together and computing the capability edges between these sets of objects as the union of the access rights between the elements of the sets. With suitable grouping of objects, this process results in Figure 3 for the SAC. The figure shows the initial system state after boot, the objects in parentheses (R) and (R-mem) are areas of memory which will later be turned into the main Router thread and its memory frames using the *create* operation, an abstraction of the seL4 system call that will create the underlying objects.

This picture now describes an abstract version of the design. We have currently not formally proved the connection between this model and the capDL specification, neither have we formally proved that the grouping of components is a correct abstraction, but it is reasonably clear that both are possible in principle.

For a formal security analysis, we first need to express the behaviour of RM in some way. In this case, we have chosen a small machine-like language with conditionals, jumps, and seL4 kernel calls as primitive operations. For all other components, we specify that at each system step, they may nondeterministically attempt any operation — it is the job of the kernel configured to the capability distribution in Figure 3 to prevent unwanted accesses.

To express the final information flow property, we choose a label-based security approach in this example and give each component an additional bit of state: it is set if the component potentially has had access to data from NIC A. It is easy to determine which effect each system operation has on this state bit. The property is then simple: in no execution of the system can this bit ever be set for NIC B.

Given the behaviour of the trusted component, the initial capability distribution, and the behaviour of the kernel, we can formally define the possible behaviours of the overall system and formally verify that the above property is true. This verification took a 3-4 weeks in Isabelle/HOL and less than a week to conduct in SPIN, although we had to further abstract and simplify the model to make it work in SPIN.

A more detailed description of this analysis has appeared elsewhere [1].

5 What is Missing?

With the analysis described so far, we do not yet have a high-assurance system. This section explores what would be needed to achieve one.

The main missing piece is to show that the behaviour we have described in a toy machine language for the security analysis is actually implemented by the 2,000 lines of C code of the Router Manager component. Most of these 2,000 lines are not security critical. They deal with setting up Linux instances, providing them with enough information and memory, keeping track of memory used etc. Getting them wrong will make the system unusable, because Linux will fail to boot, but it will not make it break the security property. The main critical parts are the possible sequence of seL4 kernel calls that the Router Manager generates to provide the Linux Router instance with the necessary capabilities to access network cards and memory. Classic refinement as we have used it to prove correctness of seL4 could be used to show correctness of the Router Manager.

Even with this done, there are a number of issues left that I have glossed over in the description so far. Some of these are:

- The SAC uses the unverified x86/IOMMU version of seL4, not the verified ARM version. Our kernel correctness proof would need to be ported first.
- We need to formally show that the security property is preserved by the existing refinement.

- We need to formally connect capDL and access control models. This includes extending the refinement chain of seL4 upwards to the levels of capDL and access control model.
- We need to formally prove that the grouping of components is a correct, security preserving abstraction.
- We need to formally prove that the user-level root task sets up the initial capability distribution correctly and according to the capDL specification of the system.
- We need to formally prove that the information flow abstraction used in the analysis is a faithful representation of what happens in the system. This is essentially an information flow analysis of the kernel: if we formalise in the analysis that a Read operation only transports data from A to B, we need to show that the kernel respects this and that there are no other channels in the system by which additional information may travel. The results of our correctness proof can potentially be used for this, but it goes beyond the properties we have proved so far.

6 Conclusion

We have demonstrated that formal code verification at a scale of about 10,000 lines of code is possible and feasible. We have argued that, for high-assurance systems, it is also cost-effective. There are no real barriers to conducting verifications like this routinely.

The bad news is that while these techniques may optimistically scale in the next few years up to 100,000s lines of code for nicely structured, appropriate code bases, realistic systems beyond that size still seem out of reach for the near future. Modern embedded systems frequently comprise millions of lines of code. None of these large systems are high-assurance systems yet, but a clear trend towards larger and more complex systems is observable even in this space, and some of these large systems, e.g. automobile code, should become high-assurance systems, because current practices are unsatisfactory [14].

Even though we may not be able to prove full functional correctness of such systems in the near future, our thesis is that it is nevertheless possible to provide formal, code-level proof of specific safety and security properties of systems in the millions of lines of code. We plan to achieve this by exploiting formally verified microkernel isolation properties, suitable security architectures, and code-level formal proofs for the small trusted computing base of such systems.

Acknowledgements 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.

This material is in part based on research sponsored by the Air Force Research Laboratory, under agreement number FA2386-09-1-4160. The U.S. Government is authorised to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

References

1. June Andronick, David Greenaway, and Kevin Elphinstone. Towards proving security in the presence of large untrusted components. In Gerwin Klein, Ralf Huuck, and Bastian Schlich, editors, *Proceedings of the 5th Workshop on Systems Software Verification*, Vancouver, Canada, October 2010. USENIX.
2. William R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.
3. William R. Bevier, Warren A. Hunt, J. Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, 1989.
4. Carolyn Boettcher, Rance DeLong, John Rushby, and Wilmar Sifre. The MILS component integration approach to secure information sharing. In *27th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, St. Paul, MN, October 2008.
5. Andrew Boyton. A verified shared capability model. In Gerwin Klein, Ralf Huuck, and Bastian Schlich, editors, *Proceedings of the 4th Workshop on Systems Software Verification*, volume 254 of *Electronic Notes in Computer Science*, pages 25–44, Aachen, Germany, October 2009. Elsevier.
6. David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 167–182, Montreal, Canada, August 2008. Springer.
7. Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Markus Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42, Munich, Germany, 2009. Springer.
8. Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. Verified protection model of the seL4 microkernel. Technical Report NRL-1474, NICTA, October 2007. Available from `http://ertos.nicta.com.au/publications/papers/Elkaduwe_GE_07.pdf`.
9. Richard J. Feiertag and Peter G. Neumann. The foundations of a provably secure operating system (PSOS). In *AFIPS Conference Proceedings, 1979 National Computer Conference*, pages 329–334, New York, NY, USA, June 1979.
10. Bill Hart. SDR security threats in an open source world. In *Software Defined Radia Conference*, pages 3.5–3 1–4, Phoenix, AZ, USA, November 2004.
11. Gerwin Klein. Correct OS kernel? proof? done! *USENIX ;login:*, 34(6):28–34, December 2009.
12. Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. *Communications of the ACM*, 53(6):107–115, June 2010.

13. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009. ACM.
14. Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental security analysis of a modern automobile. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 447–462, Oakland, CA, USA, May 2010.
15. Ihor Kuz, Gerwin Klein, Corey Lewis, and Adam Walker. capDL: A language for describing capability-based systems. In *Proceedings of the 1st Asia-Pacific Workshop on Systems*, New Delhi, India, August 2010. To appear.
16. Jia Meng, Lawrence C. Paulson, and Gerwin Klein. A termination checker for Isabelle Hoare logic. In Bernhard Beckert, editor, *Proceedings of the 4th International Verification Workshop*, volume 259 of *CEUR Workshop Proceedings*, pages 104–118, Bremen, Germany, July 2007.
17. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.
18. Raymond J. Richards. Modeling and security analysis of a commercial real-time operating system kernel. In David S. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 301–322. Springer–Verlag, 2010.
19. Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 97–108, Nice, France, January 2007.
20. Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, 1980.
21. Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. Mind the gap: A verification framework for low-level C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 500–515, Munich, Germany, August 2009. Springer.
22. Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 99–110, Toronto, Ontario, Canada, June 2010. ACM.

capDL: A Language for Describing Capability-Based Systems

Ihor Kuz, Gerwin Klein, Corey Lewis, Adam Walker
NICTA and University of New South Wales
Sydney, Australia
firstname.lastname@nicta.com.au

ABSTRACT

Capabilities provide an access control model that can be used to construct systems where safety of protection can be precisely determined. However, in order to be certain of the security provided by such systems it is necessary to verify that their capability distributions do in fact fulfil requirements relating to isolation and information flow, and that there is a direct connection to the actual capability distribution in the system. We claim that, in order to do this effectively, systems need to have explicit descriptions of their capability distributions. In this paper we present the capDL capability distribution language for the capability-based seL4 microkernel. We present the capDL model, its main features and their motivations, and provide a small example to illustrate the language syntax and semantics. CapDL plays a key role in our approach to development, analysis, and verification of trustworthy systems.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design; D.4.6 [**Operating Systems**]: Security and Protection; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Languages, Design

Keywords

Capabilities, capability distribution, security analysis, seL4, Isabelle

1. INTRODUCTION

Capabilities [1] are a powerful approach to building secure systems. They provide an access control model that allows system designers to minimise authority of processes and that can be used to precisely analyse the protection state of such systems. This is particularly useful in the presence of security requirements that limit information flow and impose isolation between system components. In this paper we motivate the need for, and introduce our specific approach to, explicit descriptions of the capability distribution in such systems.

In a fully capability-based system, all objects, including resources such as devices and memory, and system objects, such as processes and communication channels, are referenced by *capabilities* – unique tokens that act both as references and provide access rights to objects. In order to access an object or perform an operation on one, a subject must hold a capability to this object, and the capability must provide sufficient rights for the operation. Capabilities may be transferred between subjects, meaning that the set of objects accessible by subjects can change over time. If a subject does not possess a capability to an object, and cannot ever acquire such a capability, it will not be able to access the object.

Since objects are only accessed through capabilities, they can be used to restrict a subject's access to only those objects that the subject requires to perform its tasks correctly, but to no others. This allows systems to be designed according to the *principle of least privilege* [9]. Furthermore, the capabilities in a system can be distributed such that they create distinct isolated subsystems, where subjects in different subsystems cannot influence or communicate with each other in any way.[1] Besides strict isolation, the capability model can also be used to create systems that allow limited inter-subsystem communication over authorised channels. This enables the construction of systems in which *information flow* is strictly controlled.

A capability-based access control model that also provides a suitable authority transfer scheme, such as *take-grant* [6], can be shown to be *safe*. This means that all future access rights that a subject may obtain can be decided by analysing the current system state. Thus, in order to determine the security of a capability-based system (in particular with regard to its access and information flow policies) it is sufficient to analyse its *capability distribution*, i.e., the distribution of capabilities over all the subjects in the system. Such an analysis will take into account all possible transformations of the capability distribution to identify subsystems and the possible information flow between them. Given specific access and information flow requirements, the analysis can be used to determine whether a system successfully fulfils these requirements [2].

In existing capability-based systems the capability distributions are implicitly defined by the code that creates objects and transfers capabilities between subjects at runtime. A security analysis thus requires that the code first be analysed to determine which capabilities exist, how they are initially set up, and how they are subsequently propagated throughout the system. Depending on the code, this could be a complex, and potentially infeasible, process.

We propose that capability-based systems should have an explicit representation of the system's capability distribution. While having such a representation is essential for performing a security analy-

APSys 2010, August 30, 2010, New Delhi, India.

[1] Providing the system does not contain any side channels that bypass the capability-based access control.

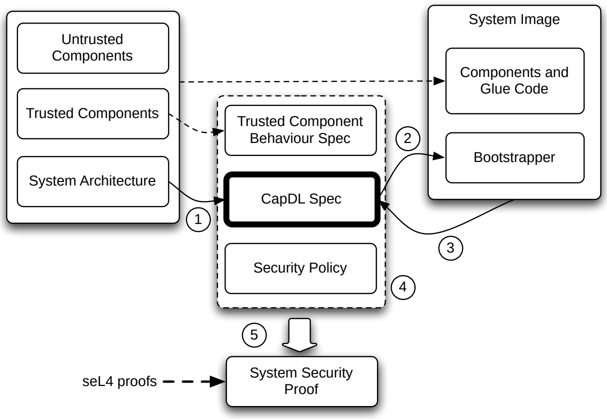

Figure 1: The role of capDL in development, analysis, and verification.

sis or formal security verification of the system, it provides other useful benefits. A clear description of a system's desired capability distribution helps the design and implementation process as well as debugging and documentation. Debugging of the system is facilitated by being able to refer to the expected capability distribution, as well as by having access to the actual capability distribution of the system being debugged. A description of a system's expected and actual capability distribution can be used to produce clearer and more complete system documentation.

In this paper we present *capDL*, a capability distribution language for seL4 [5], a formally verified capability-based microkernel. The purpose of capDL is to describe state snapshots of systems running on seL4.

CapDL plays a central role in our overall vision for development, analysis and verification of trustworthy embedded systems (Figure 1). In this vision, the capDL specification of a system is manually written or generated from a system architecture description (1). The specification can be combined with component code, glue code and a bootstrapping process to produce a runnable system image (2). Alternatively a capDL specification can be dumped from a running system (3). The capDL specification (whether hand-written, generated, or dumped) together with behaviour specifications of the system components serve as input into a security analysis tool that verifies whether the system architecture fulfils the required security policy (4). This verification can be further extended with refinement proofs of the underlying seL4 kernel to prove security properties of the actual system implementation (5).

In the rest of this paper we first present a brief overview of seL4 in Section 2 followed by a description of the capDL language in Section 3 and a comparison of capDL with other approaches to describing capability-based systems in Section 4. CapDL is a work in progress. In Section 5 we present the current status of this work and discuss our plans for its future. Finally we conclude in Section 6.

2. OVERVIEW OF SEL4

The seL4 microkernel is a small operating system kernel designed to be a secure, safe, and reliable foundation for a wide variety of application domains. As a microkernel, it provides a minimal number of services to applications. The kernel services are general enough for composing more complex operating system services that run as applications on the microkernel. In this way, the functionality of the system can be extended without increasing the code and complexity in privileged mode, while still supporting a wide number of services for varied application domains

Kernel services are provided through a small set of kernel implemented objects whose methods can be invoked by applications. These objects can only be accessed and manipulated using tamper-proof capabilities. The operations an application can perform are, therefore, determined by the set of capabilities the application possesses. The capabilities are stored in kernel managed memory and can only be manipulated indirectly through the kernel. Capabilities can be copied, moved, and sent using seL4's inter-process communication (IPC) mechanism. The propagation of capabilities through the system is controlled by a take-grant-based model.

The set of objects implemented by the kernel can be grouped into six categories:

Capability Management Capabilities in seL4 are stored in kernel-protected objects called *CNodes*. A CNode has a fixed number of slots which is determined when the CNode is created. Individual CNodes can be composed into a *CSpace*, a set of linked CNodes. In order to invoke an operation on a capability, that capability must be stored in an application's CSpace.

Object and Memory Management The *Untyped Memory* capability is the foundation of memory allocation and object creation in the seL4 kernel. A kernel object is created by invoking the *retype* method on an Untyped Memory capability. After a successful retype invocation a capability to the new object is placed in the application's CSpace. Untyped capabilities can also be used to reclaim retyped memory with the *revoke* method.

Virtual Address Space Management A virtual address space in seL4 is called a *VSpace*. In a similar way to CSpaces, a VSpace is composed of objects provided by the microkernel. The objects for managing virtual memory are architecture specific. On the Intel IA32 architecture the root of a VSpace consists of a *Page Directory* object, which contains references to *Page Table* objects, which themselves contain references to *Frame* objects representing regions of physical memory.

Thread Management Threads are the unit of application execution in seL4 and are scheduled, blocked, unblocked etc, depending on the application's interaction with other threads. A *TCB* (thread control block) object exists for each thread and provides the access point for controlling the thread. A TCB contains capabilities that define the thread's CSpace and VSpace. Note that multiple threads can share the same CSpace and VSpace or parts thereof.

Inter-process Communication (IPC) *Endpoints* (EP) are used to facilitate inter-process communication between threads. *Synchronous Endpoints* provide rendezvous-style communication, allowing the passing of data or capabilities between applications. When only notification of an event is required (with no need to send message data), then *Asynchronous Endpoints* (AEP) can be used.

Device I/O management Device drivers run as applications outside of the microkernel. To support this, seL4 implements I/O specific objects that provide access to I/O ports, interrupts, and I/O address spaces for DMA-based memory access.

3. THE CAPDL LANGUAGE

The main purpose of capDL is to describe the capability distribution of a system running on top of seL4. The language is intended to be used in several scenarios and has been designed with these in mind. Initially two separate goals led to the development of the language.

The first goal was to have a representation of the system that was suitable for security analysis, which would involve mapping a capability distribution to a security model and determining whether it fulfils security requirements. The second goal was to enable developers to easily specify the desired capability distribution of their system and provide it as input to a bootstrapping process, which would automatically create required objects and configure and populate the appropriate spaces to reflect the specified structure.

For the first goal what is needed is a format for describing a snapshot of the capability distribution in a system. To be suitable for security analysis the snapshot must describe which objects exist in the system and which capabilities they have access to. For the second goal the specification must be sufficiently detailed to allow automated code generation. Therefore it needs to include all information about capability arguments that such implementations will need. Some of this information will not be relevant for security analysis. For instance, for a security analysis it is necessary to know which frames of physical memory a process in the system can access via its virtual memory, but it is not necessary to know under which virtual address each of these physical addresses is visible to the process. For a concrete implementation of a bootstrapping component on the other hand, this latter information is crucial.

CapDL allows specifications to be underspecified, that is, to omit details of objects or capabilities, or whole objects themselves. A key use for underspecification is in early design and for communication of system designs. For example, a specification may omit objects required for bookkeeping during system initialisation, since these are not necessary to understand the overall system. CapDL also allows abstraction of specification, which involves creating a new specification that contains less details, but has equivalent semantics. For example, to facilitate security analysis we can often abstract a complex CSpace graph into a single CNode containing all relevant capabilities — reducing the complexity of analysing a hierarchy of CNodes, but maintaining the semantics of the original CSpace.

Besides these, the language has several other requirements. For developers writing manual specifications it is important that system specifications are easy to write. Thus the language provides shorthand for parts that are tedious to enter manually. For example, in capDL, large contiguous blocks of untyped memory objects and capabilities can be specified as ranges, rather than separate entries for each individual object and capability. Likewise commonly used objects and capabilities can be given meaningful names so that specifications can act as documentation and reflect intentions as well as structure. On the other hand, for specifications that are generated automatically (for example when dumping the state of the system), shorthand is not appropriate, so the language also has the option of representing a capability distribution in the most straightforward way, without requiring use of more complex shorthand. Of course, specifications differing only in the use of shorthand should be equivalent, and in capDL it is possible to show their equivalence.

A capDL specification has two main sections, the *objects* section, specifying all the objects in the system, and the *capabilities* section, specifying all the capabilities in the system.

The language model reflects the seL4 kernel object model as described in Section 2. All seL4 object types are supported by capDL. Since some types are architecture specific, each capDL specification must include an architecture declaration, which subsequently limits the object types that may be used in it.

Capabilities are typed based on the object type that they reference. A capDL capability includes a reference to the object that it refers to, and, if applicable, capabilities also include a field for the access rights that the capability bestows. Besides access rights, some capability types store extra information relating to how the object is accessed. For example, Endpoint capabilities also store a *badge* that is sent during IPCs performed through that capability. These badges help to identify participants in the communication occurring over the endpoint.

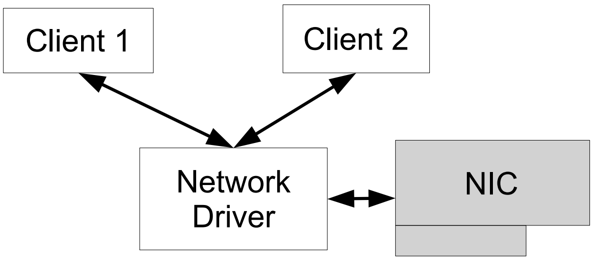

Figure 2: An example seL4-based system.

There are two main classes of objects in capDL: those that are containers, which store capabilities, and those that are not. Container objects include TCBs, CNodes, Page Directories (PD), and Page Tables (PT). These objects provide a mapping from slots to capabilities. By default slots are numbered, but can also be explicitly named to improve clarity. Some containers, such as TCBs, have a fixed size, while others can be created with arbitrary sizes.

Untyped Memory objects are also containers, but are different to the others in that they do not store capabilities, but are conceptual containers for other kernel objects. When a new object is created by retyping an existing Untyped Memory object, the new object is contained in (or *covered* by) that Untyped Memory object. Since Untyped Memory objects can be retyped into smaller Untyped Memory objects, hierarchies of these objects can exist.

Non-container objects are always of a fixed size and include Endpoints and Frames. Note that, while Frames can be of different sizes, these sizes are limited by the architecture, and different sized Frames are conceptually separate object types.

We illustrate the capDL language using a simple example system shown in Figure 2. This system consists of three components: a driver component that has access to a network interface (NIC) device and two client components that communicate with the driver. Each component runs as a separate process and has its own protected address space and individual CSpace.

Figure 3 shows a more detailed view of the objects and capabilities used in the system For clarity we show only the network driver and one client. We see that each component runs a single thread and therefore contains a single TCB. Each component has a single-CNode CSpace and a small virtual address space in which to run. The components have access to endpoints over which they communicate, and they both share a small region of virtual memory which they use to transfer packet data. The network driver component also has access to NIC interrupts through an asynchronous endpoint.

A fragment of the corresponding capDL specification is shown in Figure 4. This specification starts with the *objects* section (line 1), which lists all the objects used in the system. Note that all the objects belonging to a single component are derived from the same Untyped object (lines 3, 12, and 13). This is not required, but makes it easy to destroy and clean up after a process by revoking the capability to the parent Untyped object and making the children inaccessible.

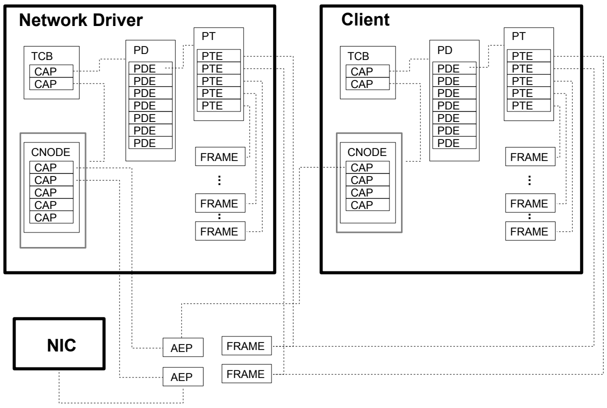

Figure 3: Capabilities involved in the example system.

```
objects -- The object section starts here                 1
                                                          2
DRIVER_ut = ut {                                          3
 DRIVER_tcb = tcb                                         4
 DRIVER_cspace = cnode (10 bits)                          5
 DRIVER_code[5] = frame (4k)                              6
 DRIVER_data[11] = frame (4k)                             7
 DRIVER_vspace = pd                                       8
 DRIVER_pt = pt                                           9
}                                                         10
                                                          11
CLIENT1_ut = ut {...}                                     12
CLIENT2_ut = ut {...}                                     13
                                                          14
DRIVER_aep = aep                                          15
IRQ_aep = aep                                             16
SHARED_frames[2] = frame (4k)                             17
                                                          18
caps -- The capabilities section starts here              19
                                                          20
DRIVER_tcb = {                                            21
  cspace: DRIVER_cspace                                   22
  vspace: DRIVER_vspace                                   23
}                                                         24
                                                          25
DRIVER_cspace = {                                         26
  1: DRIVER_aep (RW, badge: 0)                            27
  2: IRQ_aep (R)                                          28
}                                                         29
                                                          30
CLIENT1_cspace = { 1: DRIVER_aep (RW, badge: 1) }         31
CLIENT2_cspace = { 1: DRIVER_aep (RW, badge: 2) }         32
DRIVER_vspace = { 1: DRIVER_pt }                          33
                                                          34
DRIVER_pt = {                                             35
  0x0: SHARED_frames[] (RW)                               36
  0x2000: DRIVER_code[] (R)                               37
  0x7000: DRIVER_data[] (RW)                              38
}                                                         39
```

Figure 4: A capDL specification.

```
DRIVER_tcb = {
  : DRIVER_aep (RW, badge: 0)
  : IRQ_aep (R)
  : SHARED_frames[] (RW)
}
```

Figure 5: An abstraction of a capDL specification.

This is a common pattern in seL4-based systems, and being able to specify it is an important feature of the language.

The capability distribution is described in the *capabilities* section (line 19) where we place capabilities to the appropriate objects in the various containers. We show the network driver's TCB (line 21) and CNode (line 26), and the clients' CNodes (lines 31 and 32). Note that the CNodes contain capabilities to the shared endpoints, but that the capabilities have different badges (lines 27, 31, and 32). This distinguishes the two clients from each other when they communicate with the driver. We also show the structure of the network driver's VSpace (lines 33 and 35) consisting of a Page Directory, a Page Table and various Frames. While not shown here, shared memory is created by mapping the same Frame objects into different VSpaces.

While this example has been kept small in order to keep it simple, it nevertheless highlights some of the key features of the language that fulfil our requirements. As we've mentioned, it allows shorthand for ease of writing and reading (for example, in lines 6 and 7 we specify multiple objects in a single statement, then refer to these objects in lines 37 and 38). It also allows underspecification and abstraction of capability distributions. In lines 3 to 10, for example, we leave out details of Untyped object hierarchies: an implementation of this distribution may actually use a hierarchy of Untyped objects instead of a single Untyped object to create these objects. This is useful both for system description and for analysis.

We extend this example to show how capDL supports abstraction for system analysis. The initial specification shows a detailed process description consisting of a TCB, a CSpace, and a VSpace (lines 21 to 23). Figure 5 represents an abstract version of this process consisting of a single TCB that contains all externally accessible capabilities (the AEPs for communication and interrupts as well as the Frames shared with the client processes). Such a TCB is not a valid seL4 object, however, it may be a valid abstraction of a seL4 process (if refinement can be proved) and can simplify reasoning about such processes. Given that capDL has formally defined semantics, we intend to explore the automatic generation of such abstractions. Note that capDL-based system abstraction and analysis is still a work in progress, and a full discussion of this is, therefore, outside the scope of this paper.

4. RELATED WORK

Existing operating systems with capability-based access control such as KeyKOS [4], EROS [12], and Amoeba [8] do not provide means to explicitly define capability distributions. Capabilities are distributed by the system code at runtime and thus distribution remains implicit in the code. Coyotos [10] introduced CapIDL [11] which is a CORBA IDL based language for describing IPC interfaces of the processes in the system. While the CapIDL interfaces are related to, and represent, capabilities, the language does not provide a means for explicitly describing which processes provide and use which interfaces, so the actual capability distribution is never made explicit. Language-based capability systems such as E [7] provide a way to describe capabilities in a programming language (typically as references to objects), however, as with capability-based operating systems the capability distribution is implicit in the code, and is never explicitly presented as with capDL. Higher level architecture description languages such as AADL [3] do provide a means of describing an overall system architecture including the components and their interconnections, however, the abstractions that they operate on are at a much higher level than a capability distribution. Such architecture descriptions could potentially be mapped down to a capDL system description given appropriate mappings between the high-level concepts and the capabilities required to implement them. Various formal models of capability systems exist such as the original take-grant model [6]. Their purpose is a formal security analysis or specification. They are lacking the necessary detail for system implementation and debugging tools. The advantage of our approach is the direct connection between analysis and implementation within one language. The textual capDL language presented here is one way of representing the underlying model. We have also developed a binary capDL representation that is used as input to our system bootstrapper. While it would be possible to use other languages such as XML to represent capDL specifications, we have not yet investigated these options.

5. STATUS AND FUTURE WORK

The capDL model has formal semantics in the theorem prover Isabelle/HOL and we have implemented a compiler for the capDL language. Besides checking syntactic correctness and consistency of a specification, the compiler also produces a canonical representation of the specification, allowing us to check the equivalence of specifications that use shorthand. The compiler is flexible and includes different backends, including one that produces a binary representation of the specification that can be used as input into a system bootstrapper, and one that produces a graphical representation of the capability distribution in the *dot* format. We also have a debugging tool that produces a capDL dump of a running system's capability distribution. We have implemented an automated bootstrapper that takes as input the binary variant of capDL and produces an appropriate capability distribution in a running system.

A new feature that we plan to introduce extends the language to allow us to specify CSpace manipulation operations (such as creating, moving and destroying capabilities). These operations will have a formal semantics and will be used for analysis of system behaviour. We are also working on integrating capDL into our overall development and security analysis process as described in Section 1. This will include a mapping of capDL to the current security model, as well as automated security analysis tools based on capDL descriptions.

6. CONCLUSIONS

CapDL is a language for explicitly describing the capability distribution of a seL4-based system. It plays a key role in our effort to design, build, analyse, and formally verify trustworthy embedded systems, tying together work being done by system developers and formal methods practitioners. We have designed capDL to closely reflect the seL4 model, and to be flexible enough to allow full and partial specification of a seL4 capability distribution. CapDL has also been designed to be easy to write by system designers as well as to be automatically generated and processed by debugging tools. The underlying capDL model has a formal semantics, which makes it suitable as a key element in our system analysis and verification tool chain. We have developed tools to process capDL specifications, generate running systems from it, dump system state to it, and have started work on developing security analyses based on capDL system descriptions.

Acknowledgements

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.

7. REFERENCES

[1] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *CACM*, 9:143–155, 1966.

[2] D. Elkaduwe, G. Klein, and K. Elphinstone. Verified protection model of the seL4 microkernel. In J. Woodcock and N. Shankar, editors, *VSTTE 2008*, volume 5295 of *LNCS*, pages 99–114. Springer, 2008.

[3] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical Report CMU/SEI-2006-TN-011, Feb 2006.

[4] N. Hardy. KeyKOS architecture. *ACM Operat. Syst. Rev.*, 19(4):8–25, Oct 1985.

[5] 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. seL4: Formal verification of an OS kernel. In *22nd SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.

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

[7] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, May 2006.

[8] S. J. Mullender and A. S. Tanenbaum. The design of a capability-based distributed operating system. *The Comp. J.*, 29:289–299, 1986.

[9] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. IEEE*, 63:1278–1308, 1975.

[10] J. Shapiro, M. S. Doerrie, E. Northup, and M. Miller. Towards a verified, general-purpose operating system kernel. In *NICTA workshop on OS verification*, Sydney, Australia, Oct 2004.

[11] J. Shapiro and M. Miller. *CapIDL Language Specification*. Johns Hopkins University, Feb 2006.

[12] J. S. Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, 1999.

seL4: Formal Verification of an Operating-System Kernel

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

ABSTRACT

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

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

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

1. INTRODUCTION

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

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

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

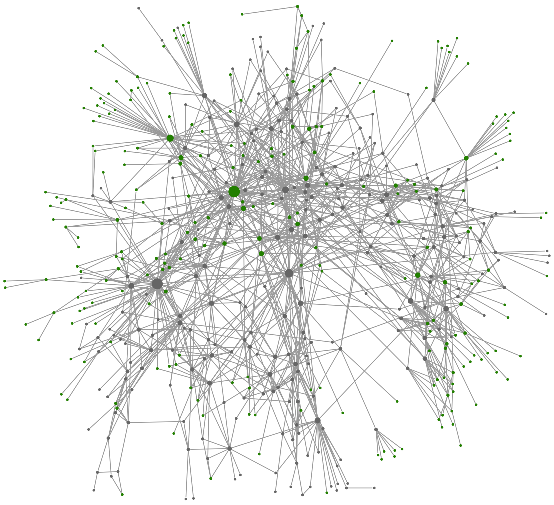

Figure 1: Call graph of the seL4 microkernel. Vertices represent functions, and edges invocations.

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

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

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

sign, an OS microkernel is entirely within the realm of full formal verification.

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

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

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

2. IMPLICATIONS

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Algorithmic invariants. Many optimisations rely on certain properties being always true, so specific checks can be left out or can be replaced by other, more efficient checks. A simple example is that the distinguished idle thread is always in thread state *idle* and therefore can never be blocked or otherwise waiting for I/O. This can be used to remove checks in the code paths that deal with the idle thread.

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

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

A verification like this is not an absolute guarantee. The key condition in all this is *if the assumptions are true.* To attack any of these properties, this is where one would have to look. What the proof really does is take 7,500 lines of C code out of the equation. It reduces possible attacks and the human analysis necessary to guard against them to the assumptions and specification. It also is the basis for any formal analysis of systems running on top of the kernel or for further high-level analysis of the kernel itself.

3. KERNEL DESIGN FOR VERIFICATION

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

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

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

3.1 Kernel/Proof Co-Design Process

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

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

3.2 Design Decisions

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

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

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

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

Kernel Memory Management. The seL4 kernel uses a model of memory allocation that exports control of the in-kernel allocation to appropriately authorised applications. While this model is mostly motivated by the need for precise guarantees of memory consumption, it also benefits verification. The model pushes the policy for allocation outside the kernel, which means we only need to prove that the mechanism works, not that the user-level policy makes sense. The mechanism works if it keeps kernel code and data structures safe from user access, if the virtual memory subsystem is fully controlled by the kernel interface via capabilities, and if it provides the necessary functionality for user level to manage its own virtual memory policies.

Obviously, moving policy into userland does not change the fact that memory-allocation is part of the trusted computing base. It does mean, however, that memory-allocation can be verified separately, and can rely on verified kernel properties.

The memory-management model gives free memory to the user-level manager in the form of regions tagged as *untyped.*

The memory manager can split untyped regions and re-type them into one of several kernel object types (one of them, *frame*, is for user-accessible memory); such operations create new capabilities. Object destruction converts a region back to untyped (and invalidates derived capabilities).

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

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

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

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

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

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

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

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

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

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

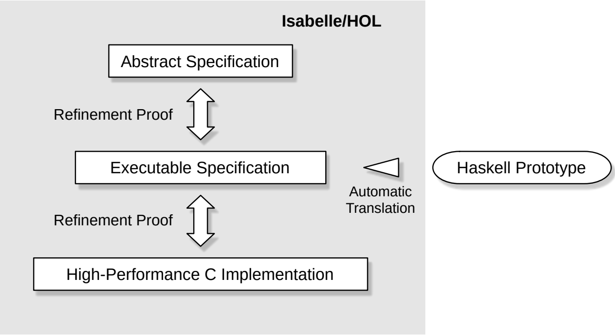

Figure 2: The refinement layers in the verification of seL4

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

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

4. VERIFICATION OF SEL4

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

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

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

4.1 Abstract specification

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

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

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

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

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

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

4.2 Executable specification

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

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

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

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

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

Figure 4: Haskell code for `schedule`.

4.3 C implementation

The most detailed layer in our verification is the C implementation. The translation from C into Isabelle is correctness-critical and we take great care to model the semantics of our C subset precisely and foundationally. *Precisely* means that we treat C semantics, types, and memory model as the C99 standard [4] prescribes, for instance with architecture-dependent word size, padding of structs, type-unsafe casting of pointers, and arithmetic on addresses. As kernel programmers do, we make assumptions about the compiler (GCC) that go beyond the standard, and about the architecture used (ARMv6). These are explicit in the model, and we can therefore detect violations. *Foundationally* means that we do not just axiomatise the behaviour of C on a high level, but we derive it from first principles as far as possible. For example, in our model of C, memory is a primitive function from addresses to bytes without type information or restrictions. On top of that, we specify how types like `unsigned int` are encoded, how structures are laid out, and how implicit and explicit type casts behave. We managed to lift this low-level memory model to a high-level calculus that allows efficient, abstract reasoning on the type-safe fragment of the kernel. We generate proof obligations assuring the safety of each pointer access and write. They state that the pointer in question must be non-null and of the correct alignment. They are typically easy to discharge. We generate similar obligations for all restrictions the C99 standard demands.

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

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

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

Figure 5: C code for part of the scheduler.

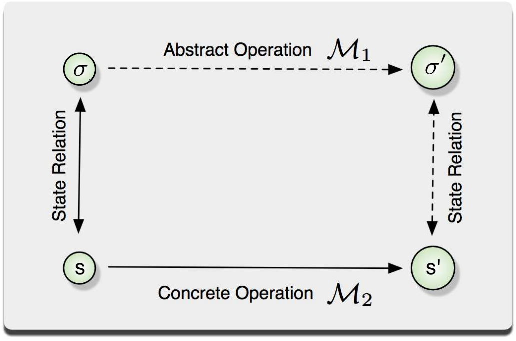
Abstract Operation $\mathcal{M}_1$
σ
σ'
State Relation
State Relation
s
s'
Concrete Operation $\mathcal{M}_2$

Figure 6: Forward Simulation.

reference parameters as much as possible, and where they were needed, used pointers to global variables (which are not restricted).

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

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

We did not use unions directly in seL4 and therefore do not support them in the verification (although that would be possible). Since the C implementation was derived from a functional program, all unions in seL4 are tagged, and many structs are packed bitfields. Like other kernel implementors, we do not trust GCC to compile and 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.

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

4.4 The proof

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

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

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

We now describe the instantiation of this framework to the seL4 kernel. We have the following types of transition in our state machines: kernel transitions, user transitions, user events, idle transitions, and idle events. *Kernel transitions* are those that are described by each of the specification layers in increasing amount of detail. *User transitions* are specified as non-deterministically changing arbitrary user-accessible parts of the state space. *User events* model kernel entry (trap instructions, faults, interrupts). *Idle transitions* model the behaviour of the idle thread. Finally, *idle events* are interrupts occurring during idle time; other interrupts that occur during kernel execution are modelled explicitly and separately in each layer of Fig. 2.

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

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

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

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

Therefore, because refinement is transitive, we have

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

4.5 Assumptions

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

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

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

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

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

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

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

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

Table 1: Code and proof statistics.

5. EXPERIENCE AND LESSONS LEARNT

5.1 Verification effort

The project was conducted in three phases. First an initial kernel with limited functionality (no interrupts, single address space and generic linear page table) was designed and implemented in Haskell, while the verification team mostly worked on the verification framework and generic proof libraries. In a second phase, the verification team developed the abstract spec and performed the first refinement while the development team completed the design, Haskell prototype and C implementation. The third phase consisted of extending the first refinement step to the full kernel and performing the second refinement. The overall size of the proof, including framework, libraries, and generated proofs (not shown in the table) is 200,000 lines of Isabelle script.

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

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

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

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

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

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

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

5.2 The cost of change

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

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

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

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

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

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

6. CONCLUSIONS

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

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

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

Acknowledgements

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

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

7. REFERENCES

[1] E. Alkassar, N. Schirmer, and A. Starostin. Formal pervasive verification of a paging mechanism. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 109–123. Springer, 2008.

[2] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *CACM*, 9:143–155, 1966.

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

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

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

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

[7] Z. Ni, D. Yu, and Z. Shao. Using XCAP to certify realistic system code: Machine context management. In *20th TPHOLs*, volume 4732 of *LNCS*, pages 189–206, Kaiserslautern, Germany, Sep 2007. Springer.

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

[9] T. Ormandy and J. Tinnes. Linux null pointer dereference due to incorrect proto_ops initializations. http://www.cr0.org/misc/CVE-2009-2692.txt, 2009.

[10] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. IEEE*, 63:1278–1308, 1975.

[11] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A fast capability system. In *17th SOSP*, pages 170–185, Charleston, SC, USA, Dec 1999.