Final Report for AOARD Grant #FA2386-10-1-4105

Formal System Verification for Trustworthy Embedded Systems

June Andronick, Gerwin Klein, Toby Murray

15 September 2012

Principal Investigators: Gerwin Klein and June Andronick

Email: gerwin.klein@nicta.com.au, june.andronick@nicta.com.au
Institution: NICTA
Mailing Address: 223 Anzac Parade, Kensington NSW 2052, Australia
Phone: +61 2 8306 0578
Fax: +61 2 8306 0406

Period of Performance: 16 Jun, 2010 - 15 Sep, 2012

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.

Abstract

The aim of AOARD project #FA2386-10-1-4105 is to investigate the creation of a framework enabling the formal verification of trustworthy embedded systems, leading to formal guarantees, holding at the source code level, about the security of large, complex systems. The key to make formal verification scale to systems consisting of million lines of code lies in minimising the verification effort by combining proofs of various components of the system in a provably correct way, and by concentrating effort on trusted components. This approach is enabled by designing trustworthy systems as componentised systems running on a small, verified microkernel such as seL4 [11].

The project, running for 27 months from 16 June 2010 to 15 September 2012, consisted of three phases:

- *Base*, running for 8 months from 16 June 2010 to 15 February 2011;
- *Option 1*, running for 7 months, from 16 February 2011 to 15 September 2011; and
- *Option 2*, running for 12 months, from 16 September 2011 to 15 September 2012.

In the *Base* phase, we identified three main research areas required to achieve the end objective, namely: (a) proving component isolation, (b) proving trusted components correctness, and (c) proving the correctness of the whole-system semantics. In this phase, we also made significant progress in the isolation proof by showing that the seL4 kernel enforces *integrity*. This integrity property was a larger proof effort, in part supported by the grant. We also worked on trusted components correctness by providing an intermediate design specification for a case-study trusted component.

In the second phase (*Option 1*), we worked on the definition of the whole system semantics, by developing an informal, yet detailed plan for achieving the decomposition of the overall theorem into proofs about the various part of the kernel-based, componentised system.

In the last phase (*Option 2*), we addressed the dual property of integrity for component isolation, that is *confidentiality*. We proved confidentiality enforcement for about 98% of seL4, and explain why the remaining functions will require some changes in seL4. As for the integrity property, this larger proof was in part supported by the grant. In this last phase, we also formalised our plan for overall system proofs in an common, initial framework in the Isabelle/HOL [15] theorem prover. This work was in part additionally funded by the AOARD extension grant #FA2386-12-1-4022 [4]. The formal framework we developed takes as input the concrete implementation (translated into formal logic) of any system made of a set of components running on top of an OS kernel. It explicitly identifies and formally states all theorems required for a given property to hold about the system. It then combines, in a provably correct way, these proofs to provide a formal proof that the property holds at the source code level of the whole system.

Contents

1 Introduction **7**

2 Background **10**
2.1 seL4 . . . 10
2.2 seL4 Verification . . . 11

3 Outcomes of Base Phase **14**
3.1 Identification of main research areas . . . 14
3.2 Proof that seL4 enforces Integrity and Authority Confinement . . . 14
3.3 Formal design specification for trusted components . . . 18

4 Outcomes of Option 1 Phase **26**
4.1 States, Transitions, System Execution . . . 28
4.2 Verification approach . . . 30
4.3 Trusted component execution up to a kernel call, without interrupt (case 1) . . . 30
4.4 Kernel call on behalf of a trusted component (case 2) . . . 32
4.5 Single untrusted component transition (case 3) . . . 33
4.6 Kernel call or interrupt when untrusted component is running (case 4) . . . 34
4.7 Trusted component execution up to a kernel call, with interrupts (case 5) . . . 36

5 Outcomes of Option 2 Phase **40**
5.1 Noninterference and Confidentiality . . . 40
5.1.1 Overview . . . 40
5.1.2 Noninterference for Operating System Kernels . . . 41
5.1.3 Confidentiality for Nondeterministic State Monads . . . 43
5.1.4 Conclusion . . . 45
5.2 Formalised framework for whole system verification . . . 46

6 Conclusion and Future Directions **51**

7 Attachments **55**

1 Introduction

This AOARD project is part of a larger research vision of building truly trustworthy embedded systems. The aim of this vision is to provide a framework for building large, complex systems, comprising millions of lines of code, and to prove specific security and safety properties for the overall system's implementation. A first step in this vision has been to provide a formally verified microkernel basis. We have previously developed the seL4 microkernel, together with a formal proof of its *functional correctness* [11]. This means that all the behaviours of seL4 C source code are included in the high-level, formal specification of the kernel.

The key idea in tackling the verification of systems on the order of a million lines of code is to minimise the Trusted Computing Base (TCB), i.e. the amount of code that needs to be verified for a given security property to hold. Minimising the TCB can be achieved by designing the system architecture in a componentised way, where untrusted components can be isolated from trusted ones. This MILS-style [1] kind of security architecture enables us to concentrate the verification effort on the trusted components. In our case the isolation is provided by the underlying kernel seL4. In seL4, all memory, devices, and microkernel-provided services require an associated access right, a so-called *capability*, to utilise them. Untrusted components can be isolated by distributing capabilities carefully, giving the components insufficient access rights to violate the property of interest.

Previous work towards this vision was included in the previous AOARD project #FA2386-09-1-4160 [3]. It involved a concrete case study system, the formalisation and high-level security proof of an existing secure access controller (SAC), as a representative example of a large complex system with a specific property requirement. In this previous project we developed:

- a formal architecture specification, describing the capability distribution and defining the components;
- a formal security model of the trusted components behaviour;
- a formal system semantics, describing the interleaving of execution between trusted and untrusted components, at the security model level;
- a formal analysis, proving some security property at the security model level.

The aim of this current AOARD project #FA2386-10-1-4105 is to transport the security property to the implementation level of the overall system. Namely the goal is to

> *"investigate the creation of a formal overall system framework that enables the use of kernel correctness, component correctness and security analysis to conclude one formal, machine-checked theorem about the overall system on the level of its implementation."*

Specifically, the objective is to

"investigate how to integrate a number of heterogeneous formal analyses into a single framework such that the overall effort for verification of the whole system is reduced to a provably correct combination of automatic, manual, and semi automatic partial proofs."

The project consists of three phases, called *Base*, *Option 1*, and *Option 2*, all aiming at the same objective, gradually building the framework, models and proofs to achieve this objective.

The present document is a final report for the whole project, describing — after a small introduction to the seL4 microkernel and its proof of functional correctness — the outcomes of each of these phases, based on previous intermediate reports and publications. Namely, the contributions are:

- in the *Base* phase:
  - identification of three main research areas required to achieve the end objective; namely: (a) proving component isolation, (b) proving trusted components correctness, and (c) proving the correctness of the whole-system semantics.
  - a proof that the seL4 kernel enforces *integrity* (contributing to (a)), leading to a publication [18], attached. This work was in part supported by this grant, a summary of the results is presented in Section 3.2.
  - an intermediate design specification for a case-study trusted component (contributing to (b)). This is part of attached thesis document [8]. A summary of the results is presented in Section 3.3.
- in the *Option 1* phase:
  - informal, yet detailed plan for achieving the decomposition of the overall theorem into proofs about the various part of the kernel-based, componentised system (contributing to (c)). This was reported in [6] and form Section 4 of the present document.
- in *Option 2* phase:
  - a proof of confidentiality enforcement for about 98% of seL4 (contributing to (a)). Together with integrity, these two properties combine to prove that seL4 enforces the classical isolation property of (intransitive) *noninterference*. This work concentrated on defining a formulation of intransitive noninterference for seL4, with a corresponding formulation of confidentiality, and proving that the combination of integrity and confidentiality guarantee noninterference. The confidentiality proof covers all of the kernel except scheduling and interrupt handling. Confidentiality is unprovable for these remaining functions, without modifying them to act more like a traditional separation kernel. We conclude from this work that these modifications will be necessary to complete the proof of noninterference for seL4. This work, in part supported by the grant, led to a publication [14], attached, and summarised in Section 5.1.

– an initial formal framework, decomposing a targeted property into intermediate proof obligations about various parts of the system, and combining them into an overall theorem about the full system (contributing to (c)). This is in part additionally supported by AOARD grant #FA2386-12-1-4022 [4] and the work is summarised in Section 5.2.

2 Background

2.1 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. The microkernel is the only software running in the *privileged* mode (or *kernel* mode) of the processor. The privileged mode is used to protect the operating system from applications and applications from each other. Software other than the kernel runs in unprivileged mode (or *user* mode). As a microkernel, seL4 provides a minimal number of services to applications: threads, inter-process communication and virtual memory, as well as a capability-based access control. Threads are an abstraction of CPU execution that support running software. As shown in Figure 1, threads are represented in seL4 by their *thread control blocks* (TCB), that store a thread's context, virtual address space (VSpace) and capability space (CSpace). VSpaces

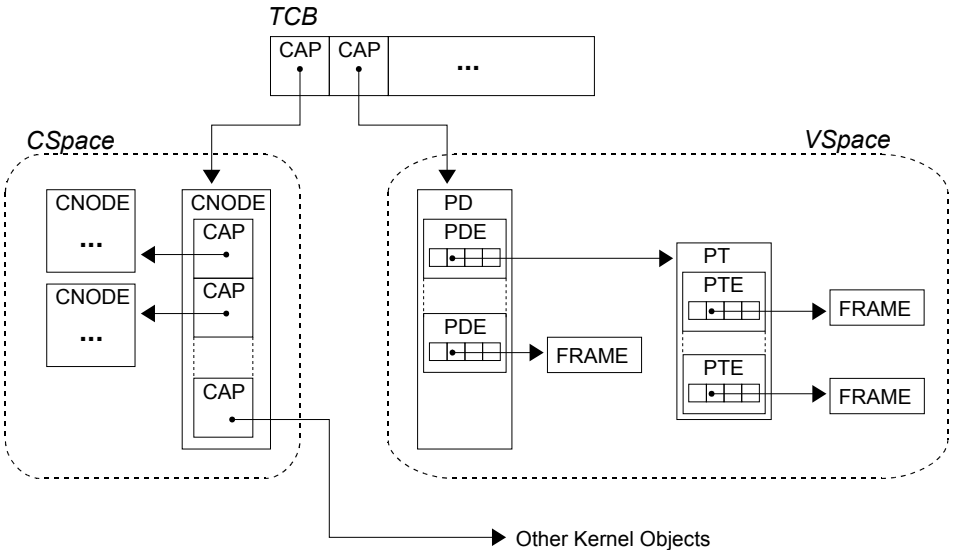


Figure 1: Internal representation of an application in seL4.

contain a set of frames, generally organised in a hierarchical, architecture-dependent structure of page tables and page directories. CSpaces are kernel managed storage for *capabilities*. A capability is an unforgeable token that confers authority. In seL4, a thread may only invoke operations on objects when it has sufficient permissions. This means that every call to the kernel requires the presentation of a capability with the correct authority to perform the specified operation. A thread stores its capabilities in its CSpace, which is a directed acyclic graph of capability nodes (CNodes). These CNodes can be of various sizes and contain capabilities to other CNodes and objects. When a thread invokes seL4, it provides an index into this structure, which is transversed and resolved into a real capability. Communication between components is enabled by *endpoints*. When a thread wants

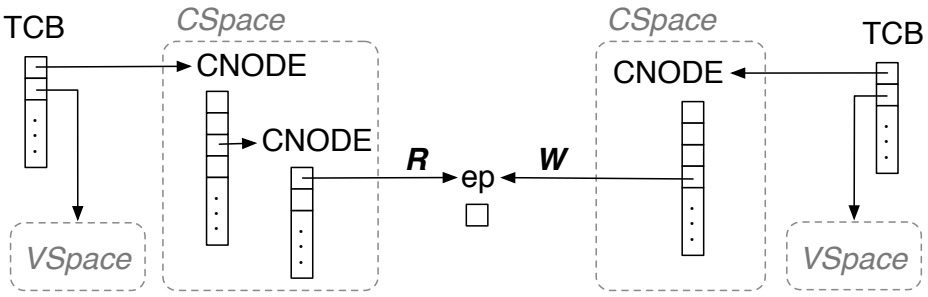


Figure 2: seL4-based system with two threads that can communicate via an *endpoint*.

to send a message to another thread, it must send it to an intermediate endpoint to which it has send access and to which the receiver has receive access, as shown in Figure 2.

The allocation of kernel objects in seL4 is performed by *retyping* an *untyped memory* region. Untyped memory is an abstraction of a power-of-two sized, size-aligned region of physical memory. Possession of capability to an untyped memory region provides sufficient authority to allocate kernel objects in this region: a user-level application can request that the kernel transforms that region into other kernel objects (including smaller untyped memory regions). At boot time, seL4 first preallocates the memory required for the kernel itself and then gives the remainder of the memory to the initial user task in the form of capabilities to untyped memory.

2.2 seL4 Verification

The seL4 microkernel was the first, and is still, to our knowledge, the only general-purpose operating system kernel that is fully formally verified for functional correctness. This means that there exists a formal, machine-checked proof that the C implementation of seL4 is a correct refinement of its functional, abstract specification. This proof assumes the correctness of the compiler, assembly code, boot code, management of caches, and the hardware. The technique used for formal verification is interactive, machine-assisted and machine-checked proof. Specifically, we use the theorem prover Isabelle/HOL [15].

As shown in Figure 3, the verification uses several specification layers. The top-most layer in the picture is the *abstract specification*: an operational model that is the main, complete specification of system behaviour. The next layer down is the *executable specification*, representing the design of the kernel. This layer is generated from a Haskell prototype of the kernel, aimed at bridging the gap between kernel development needs and formal verification requirements. Finally, the bottom layer is the high-performance C implementation of seL4, parsed into Isabelle/HOL using a precise, faithful formal semantics for the C programming language [16, 19].

The model of the machine remains the same across all refinement layers; only the details of kernel behaviour and kernel data structures change. Let machine $\mathcal{M}_A$ denote the system framework instantiated with the abstract specification, let machine $\mathcal{M}_E$ represent the framework instantiated

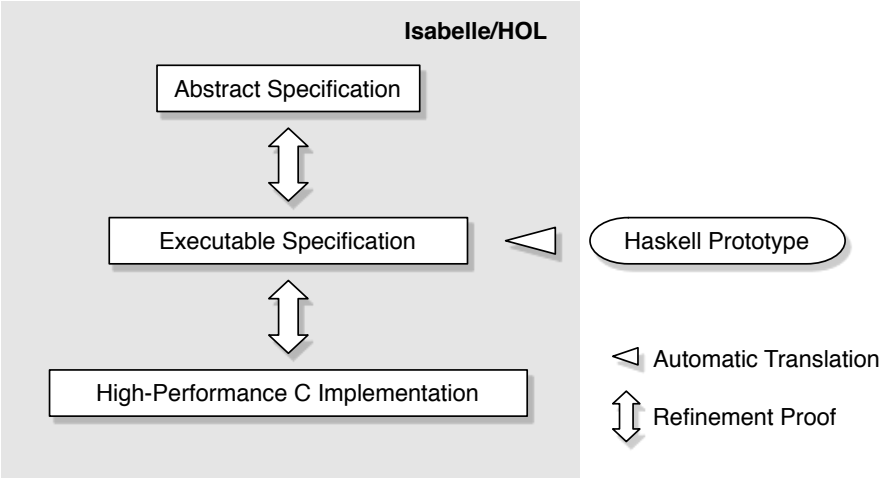


Figure 3: The refinement layers in the verification of seL4 functional correctness.

with the executable specification, 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 $\mathcal{M}_A$.*

The correspondence established by the refinement proof enables us to conduct further proofs of any *Hoare logic* properties at the abstract level, ensuring that the properties also hold for the refined model. This means that if a security property is proved in Hoare logic about the abstract model (not all security properties can be), refinement guarantees that the same property holds for the kernel source code. Since proofs at the abstract level are easier to perform, this represents a significant reduction of effort in additional verifications by over an order of magnitude, as illustrated by the proof that seL4 enforces integrity and authority confinement [18].

3 Outcomes of Base Phase

3.1 Identification of main research areas

In the *Base* phase of the project, we started by defining the key points of our approach to prove security properties at the implementation level. These are:

Formal proof of isolation. The proof of the security property at the model level is set up to ignore the behaviour of the untrusted components. It only relies on the kernel to bound their actions by the authority they have available in terms of the capabilities. This isolation provided by the kernel is not an immediate consequence of the functional correctness proof. To make sure that untrusted components can indeed be controlled by the kernel, a separate proof, building on the existing functional correctness, needs to establish that the kernel correctly enforces access control. This isolation proof can be decomposed into two properties: *confidentiality*, which means that information is not acquired without read authority, and *integrity*, which means that information is not modified without write authority.

Formal proof of refinement for the trusted components. The proof at the model level relies on a high-level model of the behaviour of the trusted components in the system. To achieve an overall implementation theorem, we need to prove that the implementation level of trusted components refines their high-level behaviour model.

Formal proof of refinement for the system semantics. To compose the separate analysis parts, we need to prove that the interleaving model used for the security analysis refines down to the implementation.

All these steps need to lead to a framework 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.

In the *Base* phase, we also made significant progress in the first two research areas: we formally proved that seL4 enforces the integrity property, which in itself represents a major research outcome; and we defined a formal specification for the main trusted component of the SAC example system studied in the previous AOARD project #FA2386-09-1-4160. These two contributions are described in detail in the next two sections.

3.2 Proof that seL4 enforces Integrity and Authority Confinement

In the area of isolation guarantees, we have proved that seL4 enforces integrity and authority confinement [18] on the implementation level. To the best of our knowledge, this result makes the seL4 kernel the first general-purpose OS-kernel with a formal, machine-checked, implementation-level proof of a high-level security property such as integrity. The effort for this proof came to less than one person year, compared to 25 person years for the previous functional correctness proof.

This demonstrates that the previous functional correctness proof can lead to a significant reduction of effort for establishing additional important kernel properties.

Access control is one of the primary security functions that OS kernels provide. It is used to enforce confidentiality and integrity of the information owned by the users of the kernel. Confidentiality means that information may not be read without read authority. Its dual, integrity, means that information may not be written without write authority.

For dynamic access control systems, such as the capability system in seL4, where a subject may transmit some of its authority to another subject, an additional property is needed: authority confinement. Authority confinement means that without explicit authority to do so, authority may not be propagated to other subjects.

In this section we summarise the proof that seL4 correctly enforces two of the above properties: integrity and authority confinement (details are published in [18]). We show why proving such properties at the source code level of a high-performance OS kernel requires more effort and a different approach than the classical approaches to analyse access control properties, such as the Bell-LaPadula model [7]. We also show how these two properties are useful for establishing bounds on the execution behaviour of user-level seL4-based systems.

An access control system controls the access of *subjects* to *objects* [12], by restricting the operations that subjects may perform on objects in each state of the system. In seL4 the subjects are threads, and the objects are all kernel objects, including memory pages and threads themselves. Proving integrity enforcement essentially comes down to proving that, for any given running thread t performing an operation o from a state s, leading to a state s', we have that s' is identical to s except for the values of the objects that t was allowed to modify in s. Similarly showing authority confinement requires to prove that, for any given object o, the authority of o in s' is not greater than its authority in s plus the authority t was allowed to propagate in s.

The part of the system state used to make access control decisions, i.e. the part that is examined by the kernel to decide if t is allowed to write or propagate authority, is called the *protection state*. In seL4, this protection state is mostly represented explicitly in the capabilities present in the capability address space (CSpace) of each subject. This explicit, fine-grained representation is one of the features of capability-based access control mechanisms. In reality, however, some implicit protection state remains, for instance encoded in the control state of a thread, or in the presence of virtual memory mappings in the MMU. The protection state governs not only what operations are allowed to be performed, but also how each subject may modify the protection state. For instance, the authority for capabilities to be transmitted and shared between subjects is itself provided by CNode and endpoint capabilities. Consider the example of two threads t_1 and t_2 communicating through an endpoint. If t_1 is the running thread, we need to analyse what is allowed to change in the state, and in particular in the objects belonging to t_2. This includes analysing what the kernel is allowed to modify on behalf of t_1 when invoked. For this, we need to consider all possible operations potentially performed on behalf of t_1. For instance, the low-level operation $set\_thread\_state\ t_2\ st$ turns the status of t_2's thread control block into st. To analyse integrity, we need to determine under which circumstances this operation is allowed to be performed and what it will modify at most. The obvious case is if t_1 has a thread capability to t_2, i.e. if the CSpace associated to the thread control

block of t_1 contains a capability pointing to the thread control block of t_2 — then t_1 is allowed to change t_2's status. However, there are other corner cases, such as if t_1 has a send capability to the endpoint, t_2 has a receive capability to the endpoint and t_2 is blocked on this endpoint. Then, if t_1 is performing a send to the endpoint, the function set_thread_state will be called to change the status of t_2 from $Blocking$ to $Running$. This should be a permitted action. What this shows is that the challenge in proving such properties for a real microkernel is that the kernel's protection state can be very detailed and cumbersome to describe formally. It is even more cumbersome to describe precisely what the allowed effects of each operation are at this level.

To address this problem, we make use of the traditional concept of a *policy* which can be seen as an abstraction of the protection state: we assign a label to each object and subject, and we specify the access rights between labels as a directed graph. The abstraction also maps the many kinds of authority in the seL4 protection state into a simple enumerated set of authority types. This simplifies the setup in three ways: the number of labels can be much smaller than the number of objects in the system, the policy is static over each system call whereas the protection state may change and, finally, we can formulate which state mutations are allowed by consulting the policy, rather than the more complex protection state. For instance, a policy for our small example of two communicating threads would associate t_1 to label A, t_2 to label B and the endpoint would get its own label. All objects contained in t_1's CSpace and virtual address space, together with t_1's TCB, would be grouped into a single label A. The simplification implies that a subject may change anything within its own label, reducing the analysis to the changes allowed in other labels. The conditions for permitted actions do not depend on the concrete state anymore, but only on the simpler policy.

The access rights between labels in our model are: Receive, SyncSend, AsyncSend, Reset, Grant, Write, Read, and Control. We make a distinction between synchronous and asynchronous send, because we are re-using the access rights in an information-flow model where the difference matters. The two other non-standard rights are Reset and Control. The former is the authority to reset an object to its initial state for re-using resources. The latter confers the authority to completely determine the behaviour of the object, e.g. by writing to a thread's registers. The Create right known from the classical take-grant models [13] maps to Control.

Using this concept, we have proved two security properties about seL4. The first property is that the predicate $pas\_refined\ p\ s$ is an invariant over all kernel calls. The predicate states that a policy p is a static over-approximation of the actual authority of the system at a given state s. Invariance of this predicate over kernel calls with the dynamically changing protection state in s means that the policy abstraction p is static and provides a static bound of authority in the system. This bound in turn means that seL4 enforces authority confinement. The second property establishes that if $pas\_refined\ p\ s$ holds for a policy p and a state s, the predicate $integrity\ p\ s\ s'$ is established after any kernel call. The predicate describes the maximum deviation of the state s' after the kernel call from the state s before the call under the policy p.

These two properties in conjunction allow us to establish bounds on the execution behaviour of untrusted user-level components by inspecting the policy only. For instance, for a suitably restrictive policy p, using the predicate $integrity\ p\ s\ s'$, we can directly conclude that all memory private

to other components of the system is equal between s and s', no matter what code the untrusted component is running or what kernel call it tried to perform. Using authority confinement, we can conclude that the policy remains the same, and that the same argument holds for the next system call. The two properties compose to ensure that untrusted components do not affect state outside their authority over whole system executions.

One important subtlety in the two properties above is that they are not true for arbitrary policies, but only if the policy satisfies certain wellformedness conditions. These conditions are similar to, but more relaxed than those of standard take-grant capability systems [13]. In particular, our wellformedness condition is subjective—it can take into account which thread is the currently running subject. This means we can make a distinction between untrusted user-level components for which we will want to make use of the security properties above and trusted ones that are verified by other means, where we do not need to make use of kernel-enforced security.

In particular, the wellformedness conditions are:

- no Control authority from the current subject to a separate security label
- no Grant access between separate security labels
- every label has full authority to itself
- if there is an interrupt with AsyncSend authority to a label l, the current subject has AsyncSend authority to a label l

The first condition would usually be phrased for all subjects, not just the current subject. The distinction allows us to have trusted components with Control authority over untrusted ones: for trusted subjects the policy is not wellformed, but it will not be needed; for untrusted subjects the policy *is* wellformed and can be applied. The second and third condition correspond to transitivity and reflexivity in the classical take-grant model. The last condition treats all interrupts as originating from the current subject; it could potentially be relaxed or phrased differently, but we have so far not found it necessary to change yet.

In summary, we have proved the following two theorems.

Theorem 4 (Integrity at abstract level)**.** *Given a policy p, an initial kernel state s and a final kernel state s', the property $integrity\ p\ s\ s'$ holds after all kernel calls at the abstract specification level, i.e. after all transitions steps of $\mathcal{M}_A$ from s to s', assuming that s refines p, assuming the general system invariants $invs$ and ct_active (current thread is active) for non-interrupt events in s, and assuming the policy subject in p is the current thread in s.*

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

Theorem 5 (Authority Confinement at abstract level)**.** *The property $pas\_refined\ p$ is invariant over kernel calls at the abstract level, i.e. over all transitions steps of $\mathcal{M}_A$, assuming again the general system invariants $invs$ and ct_active for non-interrupt events, and assuming that the current subject of the policy is the current thread.*

The proof of these two security properties was completed in the space of 4 months with a total effort of 10 person months. The proof took 10,500 lines of Isabelle script. The effort is dramatically reduced compared to the functional correctness proof, because the proof could be completed on the abstract specification instead of the code-level of seL4. However, composed with the functional correctness result, the property automatically holds for the code level as well! More precisely, using Theorem 3, we have the following two theorems at the source code level.

Theorem 6 (Integrity at code level)**.** *The property $integrity\ p\ s\ s'$ holds after all kernel calls at the source code level, i.e. after all steps of $\mathcal{M}_C$ from C states t to t', under the same assumptions as in Theorem 4, where additionally s and t as well as s' and t' are in the state relation used for the refinement proof between $\mathcal{M}_A$ and $\mathcal{M}_C$.*

Theorem 7 (Authority Confinement at code level)**.** *The property $pas\_refined\ p$ is invariant over kernel calls at the source code level, i.e. over all transitions steps of $\mathcal{M}_C$ from C states t to t', under the same assumptions as in Theorem 5, where again additionally s and t as well as s' and t' are in the state relation used for the refinement proof between $\mathcal{M}_A$ and $\mathcal{M}_C$.*

This means, while the direct formulation of $integrity$ and $pas\_refined$ only holds on abstract-level states s and s', by refinement, the image t and t' of these properties under the state relation also holds on code-level states. This image could be reformulated as a new property that only talks about t and t', but since other user-level reasoning will be performed on the abstract level as well, we have not found it necessary to do so yet.

The massive reduction in effort is the true power of the abstraction provided through the functional correctness proofs before. Any additional property that is preserved by refinement can be established more rapidly and with less effort.

The dual property to integrity, confidentiality, will be the focus of *Option 2* Phase (in Section 5).

3.3 Formal design specification for trusted components

The second main research area looks at trusted components correctness. Trusted components, by definition, can potentially violate the targeted property P that we want to prove for our given system. In other words, where untrusted components' behaviour can be completely arbitrary, assumptions need to be made about trusted components' behaviour for P to hold. For instance, in our secure access controller (SAC) example system used in AOARD report #FA2386-09-1-4160, the property (absence of data flow) was formally proved assuming some expected behaviour of the (only) trusted component of the system (and no assumption was made on untrusted components, except that the kernel prevents them from accesses not authorised by their capabilities). Transporting such security

proofs down to the implementation level requires to prove that the actual source code of the trusted component refines this high level behaviour description used for the security proof.

As experienced in our seL4 verification work (and generally acknowledged in software verification), refinement proofs from high level security requirements down to low level source code is often facilitated by the use of intermediate level of abstraction, as a functional specification or design level model. Therefore, in the case of trusted component correctness, we investigated the definition of intermediate functional specifications, with the aim of refining the high level security behaviour to this intermediate specification, and then from this latter down to the C code.

Here, we re-use our previous SAC case-study [2], whose aim was to isolate several untrusted back-end networks of different classification level, connected to a trusted front-end terminal accessing one network at a time, as illustrated in Figure 4. For simplicity and without loss of generality we will only consider two such untrusted networks.

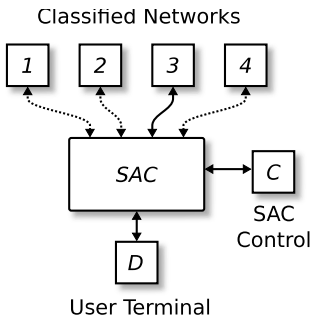


Figure 4: The SAC routes between a user's terminal and 1 of n classified networks

In this case-study, the routing task is managed by a Linux instance, carefully placed outside of the TCB, by granting it access to only one back-end network (see Figure 5). Another Linux instance, called the SAC controller, implements a web interface where the user can request a network switch. The critical task of granting the rights is performed by the only trusted component of the design, called the Router Manager.

The Router Manager sits in an infinite loop waiting for instructions from the SAC controller. Once it receives a message, it can either just tear down the Router component, or tear down the Router component and then setup a new Router to connected either network A or network B. Once the Router Manager has finished responding to the control message it will return to waiting for another message. Setting up the Router component involves creating a Router CSpace, virtual address space, inter-process communication buffer, thread control block, and then connecting them all together and activating the new thread. Regions of memory such as the Router's program text and some free space need to be mapped into the Router's virtual address space. Additionally, the capabilities to either the Network A interface or Network B interface, and some Untyped capabilities, need to be transferred into the Router's CSpace.

As already mentioned, the absence of data flow between the two back-end networks has been

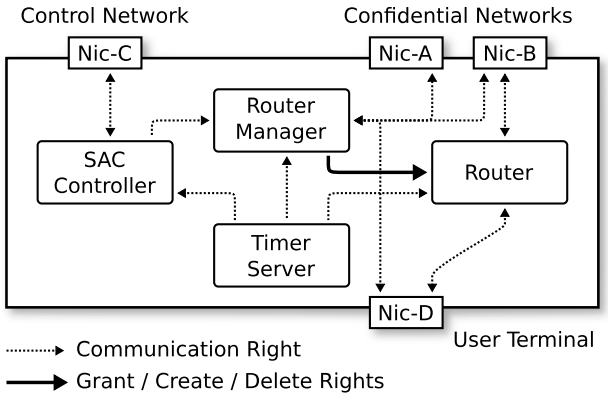


Figure 5: High level component breakdown of the SAC design

formally proved at a high level model, using a formal description of the Router Manager's desired behaviour. This proof was conducted two times with different approaches, once using model checking and once using Isabelle/HOL [2].

In this project, the aim is now to prove that the actual source code of the Router Manager refines this high-level security description. We achieved one major step in this effort by defining an intermediate, detailed monadic specification of the Router Manager component in the theorem prover Isabelle/HOL. This functional specification defines the set of all seL4 user state transitions that the Router Manager can make, at a level of abstraction similar to seL4's *abstract specification* (see Figure 6).

Before we dive into this functional specification, let us start by briefly looking at the security model of the Router Manager's behaviour (more details can be found in [8]). At this level of abstraction, all 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. Each trusted component's behaviour is modelled as a sequence 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 sys_op =
    SysRead cap | SysWrite cap bool
  | SysCreate cap | SysGrant cap cap
  | SysDelete cap | SysRemoveSet cap "(cap set)"
```

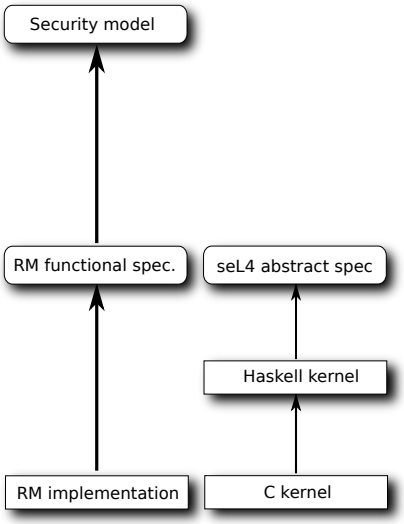


Figure 6: Side-by-side comparison of SAC and seL4 refinement artefacts.

```
datatype instruction =
      SysOp sys_op
    | Jump "nat list"
    | AnyOp

type_synonym program = "entity_id ⇒  instruction list"
```

All of the operations take a capability pointing to the targeted entity. 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
```

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.

Now the Router Manager's program is formalised in this high level security model 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),
...  ]
```

Here the capability names indicate which object they give authority to and with which right; for instance `cap_R_to_SAC_C` is defined by

```
cap_R_to_SAC_C ≡ (|entity = SacController, rights = {Read}|)
```

After the definition of this security model, the aim is now to provide a functional specification with a level of abstraction between the security model and the C implementation. Note that any user component's steps can be divided into internal processing and calls into the seL4 kernel. As we have seen, in the security model, kernel calls are represented by the abstract instruction datatype `sys_op`, and the only internal processing is the jump instructions which moves control to some other position in the program non-deterministically. In the implementation however, the Router Manager takes mostly internal steps, punctuated by kernel calls to set up or communicate with other processes. Invoking the kernel involve setting up registers and trapping into the kernel; this process is wrapped in C library functions for seL4 called libseL4. The role of the functional specification is thus to describe the Router Manager's internal steps in more detail than the security model, as well as providing a formal specification for the libseL4 functions.

We do not list the full description of the functional specification in this report; instead, details can be found in the attached thesis document [8], pages 57-60. We will just illustrate its level of abstraction by noting two interesting points. Firstly, the control flow at the functional specification level is close to both the C implementation and the high level security model, bridging the gap between the two representations. Figure 7 compares the control graphs of the C code and the functional specification, which in turn can be compared to the security model given previously in this section.

The second interesting point is that the state space at the functional specification level, on the other hand, contains much more details than just the authority. In particular, as shown in Figure 8, it represents the component's local and global variables needed to describe in more depth the internal instructions to tear down and set up new routers. The first four components are the local variables used by the (functional specification of the) Router Manager, including its instruction pointer that needs to be updated during execution. The remaining ones are global variables: *untyped_items* contains pointers to all of the *Untyped* capabilities the Router Manager has; *free_items* contains how

C code	RM Line	RM Instruction
`info= seL4_Wait(com_ep, NULL);`	0	`k_instr wait_for_message`
`message= seL4_GetMR(0);`	1	`u_instr get_message`
`switch (message) {`	2	`cnd_jmp (λ(tc,rm). message rm = 0) 4,`
`  case 0:`	3	`jmp 13,`
`    ⟨Teardown Router⟩`		`⟨Teardown Router⟩`
`    break;`	12	`jmp 86`
`  case 1:`	13	`cnd_jmp (λ(tc, rm). message rm = 1) 15,`
`    ⟨Teardown Router⟩`	14	`jmp 50`
`    ⟨Setup Router (NIC A)⟩`		`⟨Teardown Router⟩`
`    break;`		`⟨Setup Router (NIC A)⟩`
`  case 2:`	49	`jmp 86`
`    ⟨Teardown Router⟩`	50	`cnd_jmp (λ(tc, rm). message rm = 2) 52,`
`    ⟨Setup Router (NIC B)⟩`	51	`jmp 86`
`    break;`		`⟨Teardown Router⟩`
`  default:`		`⟨Setup Router (NIC B)⟩`
`    ...`		
`    break;`		
`}`		

Figure 7: Comparison of C code and Functional specification Router Manager (control flow only)

many *Untyped* capabilities of each size are free; *total_items* contains how many of each size *Untyped* the Router Manager has in total; *RM_RWGC_NIC_B_cap* contains a pointer to the Router Manager's all-rights (**R**ead **W**rite **G**rant **C**reate) capability to the network B interface; finally *client_vspace*, *virtualirq_endpoint*, and *command_endpoint* are pointers to capabilities which are set up by a system initialisation phase.

```
record sac_abstract_state =

  ip :: seL4_Word
  info :: seL4_MessageInfo
  message :: seL4_Word
  var_i :: nat

  untyped_items :: seL4_CPtr list list
  free_items :: seL4_Word list
  total_items :: seL4_Word list

  RM_RWGC_NIC_A_cap :: seL4_CPtr
  RM_RWGC_NIC_B_cap :: seL4_CPtr
  RM_RWGC_NIC_D_cap :: seL4_CPtr

  client_vspace :: seL4_CPtr
  virtualirq_endpoint :: seL4_CPtr
  command_endpoint :: seL4_CPtr
```

Figure 8: Definition of the *sac_abstract_state*

4 Outcomes of Option 1 Phase

This section reports on the results obtained during *Option 1* phase of the project. It is based on [6].

During Option 1 we have made significant progress towards transporting the overall system semantics to the implementation level by defining a formal automaton that describes the execution of the whole software system, and by providing a detailed plan for the verification of security properties by combining proofs about various parts of the system. The automaton integrates kernel transitions with known behaviour transitions of trusted components and unknown behaviour transitions of untrusted components. Untrusted component behaviour is constrained by hardware mechanisms, mainly the virtual memory subsystem, and microkernel access control. Since automaton transitions in the framework are kept as general nondeterministic relations they can be instantiated by a number of different formalisms that admit different kinds of formal analysis. The default analysis method in this framework would be interactive proof in the theorem prover Isabelle/HOL. Integration of automated provers such as first-order provers and SMT-solvers is achieved by using the existing integration of such tools with the Isabelle theorem prover (with proof re-checking in Isabelle) or by using the external tool as a trusted oracle to Isabelle (without proof re-checking). Very successful examples of such integration with Isabelle and first-order automated provers or SMT solvers can now be found in the literature [9, 10]. If a particular abstraction level and its properties manage to stay completely within the logic of any of these automatic tools, the corresponding proof can be automatic.

The remainder of this section lays out how the different parts of this framework can be put together to obtain invariant properties of system execution, reducing the reasoning necessary to reasoning about system composition, kernel access control, and trusted component behaviour.

Big picture. The aim is to be able to formally prove strong safety or security properties for large, complex software systems, typically comprising millions of lines of code. Our approach is to avoid verifying all of the code by designing the system as a set of components such that the targeted property only relies on the behaviour of a small number of components, so-called *trusted components*.

In other terms, trusted components can potentially violate the property and we need to prove that they do not—whereas untrusted components are not given enough rights to violate the property, therefore they can be as large and complex as needed and their behaviour can be arbitrary. The isolation or control of communications between components is ensured by the underlying kernel, seL4.

The kernel is the only component running in the privileged mode of the hardware, known as *kernel mode*. In this mode, the hardware allows free access to all resources. The kernel is therefore the most critical part of the system. Every other component runs in *user mode* and needs to perform a kernel call to be able to communicate with other components. User mode execution essentially only admits free access to the currently mapped memory and registers. All other interaction can be restricted via kernel access control. Which memory is currently mapped for each user component

is again controlled by the kernel and thereby subject to the security policy of the system. If a user component tries to access memory it is not authorised to access, the hardware generates a page fault which leads to a kernel call. The kernel can then react appropriately. If the user component would like to invoke any other form of communication such as interrupts, synchronous or asynchronous message passing, it needs to invoke the corresponding kernel primitive. These are again subject to the access control policy.

This strictly enforced separation by policy allows us to reason about the execution of untrusted components without a specification of their behaviour.

Limitations. The line of reasoning presented above does not include timing or other side channels of the machine that user-level components may try to use for communication. By definition, side channels are those channels that are not visible in the formal model of the system. In our case, our highest-fidelity model includes the full functional behaviour of the kernel at the C-code level as well as the memory, register, and virtual memory subsystem behaviour of the machine. It does not include caches, the TLB, or the timing behaviour of the machine. It currently also does not yet include external devices. Any of these could potentially be used as communication side channels. How far that is possible depends mainly on the hardware and the implementation of the kernel. For these we still need to rely on traditional security analyses.

In relation to devices, the model presented below does not yet take into account interrupts apart from timer interrupts. We expect to be able to handle such device interrupts within the same framework in future work.

Assumptions. Some of the reasoning in the technical section below will lead to assumptions and thereby restrictions on the system policy. That means not all policies are valid for secure systems. This is to be expected: for instance, the policy that just give everyone access to everything is not a secure policy. The main notable assumptions are that we do not permit memory sharing between security domains and make restrictions on which communication primitives are permitted between domains. In future work we are looking to relax this assumption and allow limited forms of shared memory communication. A first formal set of assumptions is detailed in our paper on the integrity proof [18]. Further policy restrictions will later arise from our work on confidentiality and system composition. Different levels of abstraction may also each bring their own restrictions with them.

Status. The status of the work described below at the end of Option 1 in the project is an initial semi-formal model and plan for formalising the rest of the framework. It is neither a fully formal model nor supported by formal proof yet. Instead it presents, at a relatively detailed level, our plan for achieving such a formalisation and our expectations on which proof obligations will remain for a full system analysis combining different levels of abstraction. The bulk of this formalisation was later completed in AOARD #FA2386-12-1-4022 and described in detail in the final report of that project [4]. Some of this work intersects with work in Option 2 which we summarise in Section 5.

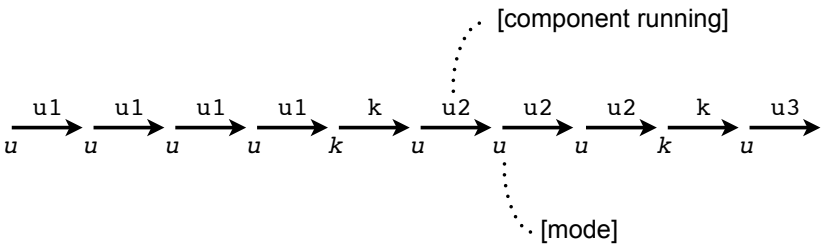


Figure 9: Example of system execution at the code level (simplified)

4.1 States, Transitions, System Execution

As mentioned, our aim is to prove that a given property holds at the source code level of a componentised system. Such a proof will generally be derived from a more general invariant about the whole system, i.e. a property I which needs to hold at each single step of any possible execution of the system. The idea is to pick an invariant that is strong enough to imply the desired security or safety property.

A typical execution of the system is represented, in a simplified version, in Figure 9: a user u_1 runs, then it performs a kernel call, which traps into kernel mode where the kernel executes the command, and then schedules another user, say u_2; this user then starts running until it in turn performs a kernel call, etc. The sequence depicted in Figure 9 is an execution trace generated by the system automaton containing the formalisation of kernel and user-component behaviour. The goal is to show properties over all possible execution traces of a given system. Each step in Figure 9 indicates which component is running, and each intermediate state indicates the current mode (user mode or kernel mode). For more precision, we will add two more pieces of information into the state. Firstly, for the beginning of each kernel execution, we note on whose behalf the kernel is running. In terms of the kernel, this is represented by the currently running thread. We do this to distinguish kernel calls of trusted components from kernel calls of untrusted components. Secondly, a kernel execution is an atomic step, as it can't be interrupted. However, a user execution can be interrupted at any time when a timer interrupt occurs. This would result in a user transition ending in a state in kernel mode, followed by a kernel transition. To represent this, we will add a *tag* indicating if the switch to kernel mode is due to a call or a timer interrupt. We currently do not yet treat other kinds of interrupts on this level, such as device interrupts, although they are already modelled in the kernel correctness proofs. Figure 10 shows our representation of states and transitions, whereas Figure 11 shows all possible transitions in any seL4-based system. Using this more detailed representation of states, the example execution given in Figure 9 would be represented as in Figure 12. Now the example where an interrupt occurs, say during the execution of user u_1, is represented in Figure 13.

So far we have illustrated the notion of system execution by example. We will keep using examples to illustrate the different theorems that will be used, but we will also try to show the general case as far as possible. In the case of execution, given the possible system transitions

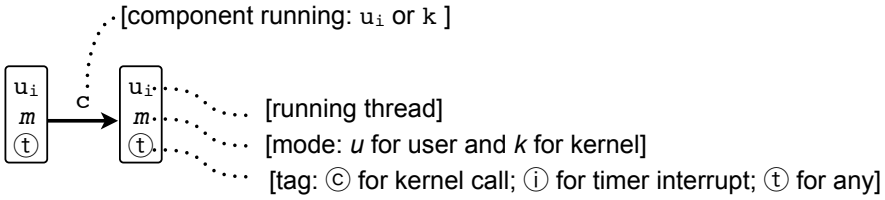


Figure 10: Representation of a system transition at the code level

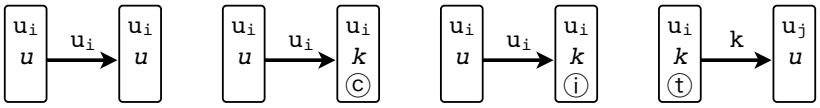


Figure 11: All possible system transitions at the code level

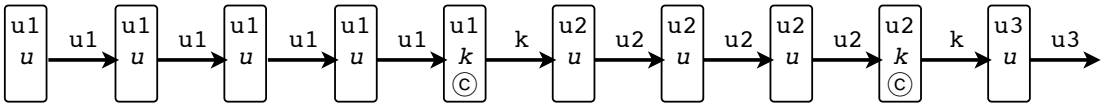


Figure 12: Example of system execution at the code level (detailed)

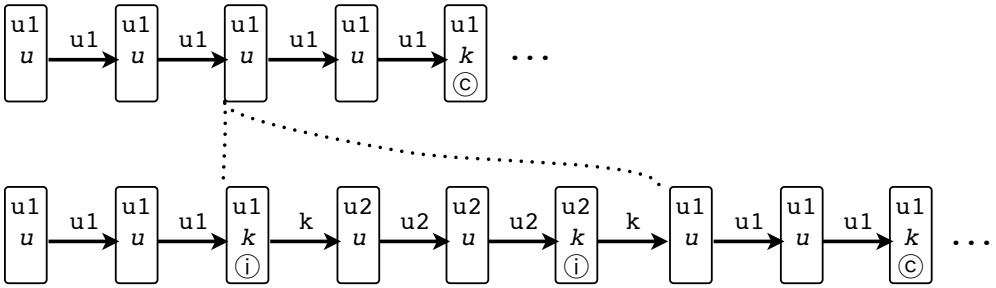


Figure 13: Example of system execution at the code level, including a timer interrupt

(Figure 11), the general definition would be as in Figure 14, i.e. an execution is a sequence of user transitions, followed by a kernel transition due to a kernel call or a time interrupt, followed by a new sequence of user transition for newly scheduled user, and so on and so forth. Figure 14 also illustrates the property we are aiming for, i.e. proving that the invariant I holds at each state of the execution.

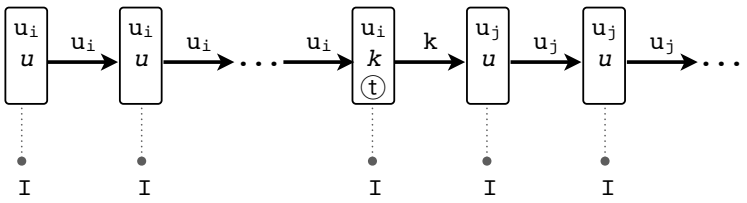


Figure 14: Representation of the general definition of a system execution, together with the property to prove, i.e. that the invariant I holds at each step of the execution.

4.2 Verification approach

The general idea for verifying that the invariant I holds at each step of the execution of the source code will be to, whenever possible, abstract the formal model of the execution at a high level where the verification will require less effort. The two main challenges in doing this will be: (1) ensure that the proof at the high level is preserved by refinement, i.e. that I holds for any concrete level implementing the abstract high level; and (2) combine possibly different levels of abstraction and different formalisms used for various parts of the execution. Indeed, a different approach will be used to deal with different chunks of the execution; for instance, reasoning about trusted components execution will require more effort than reasoning about untrusted ones.

The plan is to decompose the possible executions described in Figure 14 into the 5 following cases, illustrated in Figure 15.

1. Trusted component execution up to a kernel call, with no interrupt.
2. Kernel call on behalf of a trusted component.
3. Single untrusted component transition.
4. Kernel call on behalf of an untrusted component or timer interrupt during execution of an untrusted component.
5. Trusted component execution up to a kernel call, but interleaved with interrupts leading to execution of other components.

The verification approach is that for each of these 5 cases, we will prove that the invariant holds after each state of the execution, assuming that it held when the execution started. The approach for this proof and its level of difficulty will be different for each case. Each one is described in the following.

4.3 Trusted component execution up to a kernel call, without interrupt (case 1)

Since, by definition, the trusted components have enough rights to potentially violate the targeted property, we need to model their behaviour precisely and prove that the property is not violated by

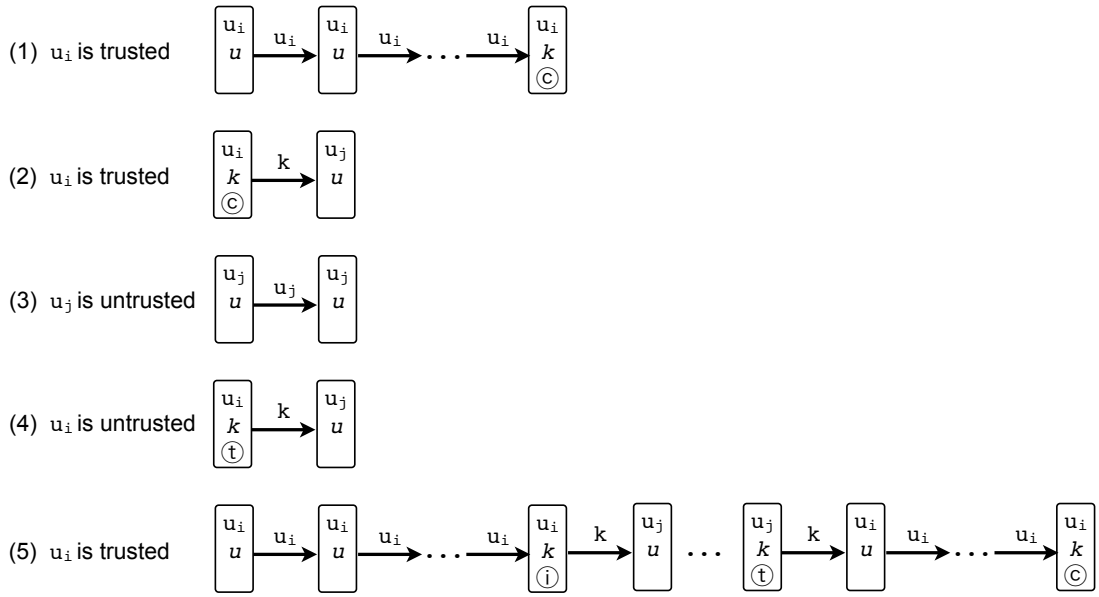


Figure 15: Decomposition of execution in 5 different cases

this behaviour. This proof can be complex and ideally we will want to conduct it at a high level of abstraction. Therefore we need to (1) model an abstraction of the behaviour of the trusted components and prove that their implementation is a refinement of this model (illustrated in Figure 16); (2) prove that the abstract version I' of the invariant I holds for the abstract model (illustrated in Figure 17); and (3) prove that if the abstracted invariant I' holds on an abstract state, then I holds on its refined state (illustrated in Figure 18). Putting everything together, we will end up in a proof that the invariant I holds at the concrete level for each state of the execution of a trusted component up to a kernel call, as illustrated in Figure 19.

We have demonstrated refinement proofs down to the source-code level before in the verification of the seL4 microkernel. An additional challenge for trusted component verification is the step atomicity. While the kernel can safely be modelled as one atomic step, because interrupts are disabled, the execution of a trusted component could be interrupted at any time and we need to show that our invariant I does not only hold at begin and end of the possibly longer abstract execution, but at the end of each atomic step on the concrete level. This means the refinement proof obligation needs to be stronger than in our previous kernel verification.

Note that during the user-level execution of trusted components, only small parts of the invariant are expected to be relevant—those that talk about the state (memory and registers) of the running trusted component. The user-level execution of a trusted component will not be able to affect any other global system state yet. For such a change to occur, the component needs to perform a kernel call which leads us into the next case.

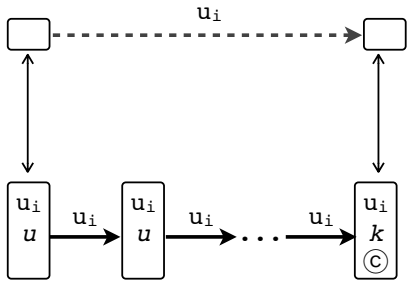


Figure 16: Refinement proof for trusted components

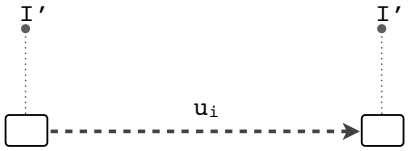


Figure 17: Invariant preservation proof at the abstract level

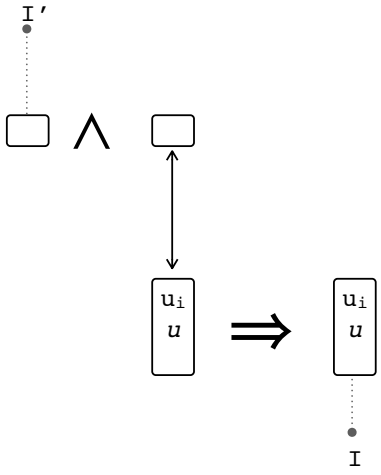


Figure 18: Proof that the invariant is preserved by refinement

4.4 Kernel call on behalf of a trusted component (case 2)

In this case, a trusted component has previously performed a computation and is about to place a kernel call. In terms of the system policy the trusted component will have enough authority for the

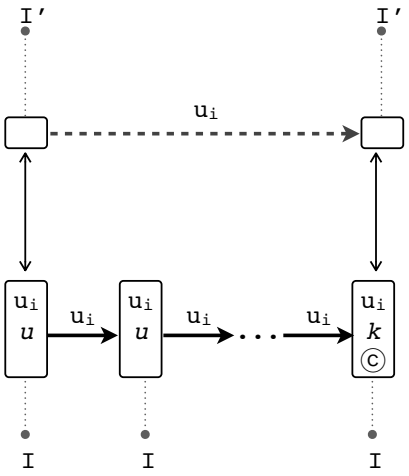


Figure 19: Proof scheme for case 1

kernel to permit actions in this call that may violate the security property and therefore the system invariant I. It is our proof obligation to show that this does not happen.

We noted in case 1 (trusted component execution) that only small parts of the invariant, if at all, will talk about the private state of the trusted component and therefore invariant preservation will not be too hard to prove. In case 2, however, the kernel call may have global effects. This is the interesting case in the invariant preservation proof. To show that these effects preserve the abstract invariant I', we need to use the result of the verification in case 1 to determine which parameters the kernel was invoked with. Given these parameters and the specification of the kernel at the suitable level of abstraction, we must show that I' is preserved. Per the existing kernel refinement theorem, we then get that I again holds after the kernel call has finished on the concrete level.

4.5 Single untrusted component transition (case 3)

Untrusted components can potentially be large and complex. The idea is not to model any details of their behaviour at all, and to just use the fact that we have restricted their authority so they cannot violate the property. The authority each component has is described as a policy graph $\mathcal{P}$. This graph describes what authority each components has to other components or to memory regions. It is an abstraction of the actual detailed protection state given by the capabilities in the system. The policy graph also indicates which is the currently running component (called *subject*). When an untrusted component, say u_j, executes a *user* transition, the only thing that it can potentially modify or read is the memory it has access to. Any other actions, such as communicating with other components, require a kernel call, which is covered by the next case. Therefore the approach here is to prove, at an abstract level, that I' does not depend on the private state of the untrusted component,

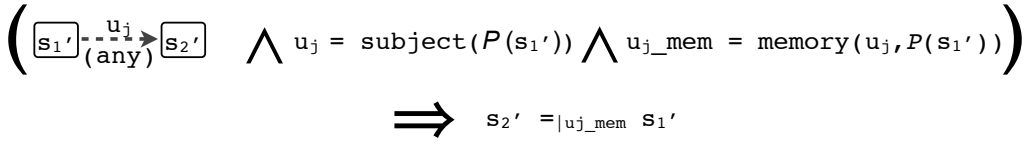


Figure 20: Resulting state for any untrusted user transition

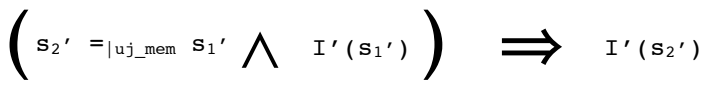


Figure 21: Invariant does not depend on untrusted user memory

in particular its memory and registers.

We do this by first proving that if s'_1 is the (abstract) state before the untrusted user transition, and s'_2 the one after, then the only thing can possible have changed in s'_2 is u_j's memory. This is noted $s'_2 =_{|u_j\_mem} s'_1$, and is illustrated by Figure 20. This fact can be proved generically and once and for all.

Then we prove that I' does not depend on u_j_mem by showing that if it holds for any given state s'_1, then it will hold on any state s'_2 that differs only on u_j_mem (as shown in Figure 21).

Now we prove that the abstract transition (modelled as *any* transition) is trivially refined by any implementation, including the actual source code of the untrusted component, as illustrated in Figure 22).

Putting all the theorems together, and using the invariant preservation by refinement (Figure 18), we obtain that the invariant I is trivially preserved by untrusted user transitions, as illustrated in Figure 23).

4.6 Kernel call or interrupt when untrusted component is running (case 4)

The same way we proved that the invariant does not depend on untrusted component executions, we need to prove that it neither depends on the kernel actions performed when those components invoke a kernel call. This case is more complicated than a user transition, because the kernel runs in privileged mode so it could potentially do more than what the running thread is authorised to do.

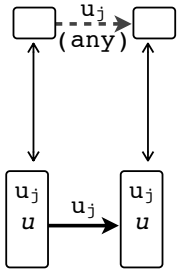


Figure 22: Refinement proof for untrusted components

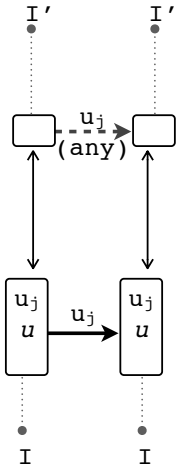


Figure 23: Proof scheme for case 3

A crucial property is therefore that it does not. The second crucial property is that the kernel call does not change the security policy. Again, the kernel potentially has the ability to do this and the proof obligation is that it does not. Both properties can be proved once and for all with the policy as a parameter. Our proof of integrity [18] does precisely this. It guarantees that the kernel will not modify objects that the running thread is not allowed to modify. In other words, if s'_1 is the (abstract) state before the kernel call — on behalf of an untrusted user u_j — and s'_2 the one after, then the only thing that is allowed to have changed in s'_2 is what u_j has write authority to as defined by the policy $\mathcal{P}$. This is noted $s'_2 =_{|u_j\text{-}W\_auth} s'_1$, and is illustrated by Figure 24.

We then need to prove that the (abstracted) invariant I' does not depend on what the untrusted components are allowed to modify, as illustrated in Figure 25.

The invariant I' is likely to mention the policy and/or the protection state of the system. We

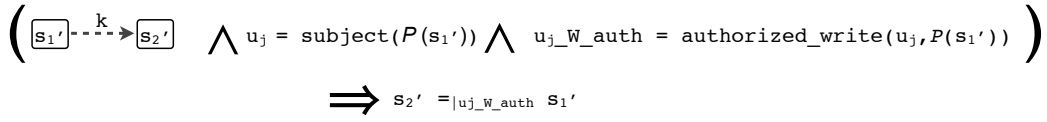


Figure 24: Integrity theorem

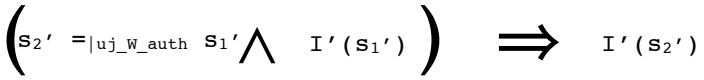


Figure 25: Invariant does not depend on what untrusted users can modify

therefore need the second property mentioned above, *authority confinement*. In particular, authority confinement implies that an untrusted component cannot change the policy graph, and therefore the policy graph is unchanged through all untrusted component transitions. This enables the theorems on such transition to be chained into longer executions.

Again, we now need to prove that the abstract kernel transition is refined by the kernel source code transition and therefore the concrete level preserves I. This is given by our previous seL4 correctness proof [11]. Putting all the theorems together, we obtain that the invariant I is preserved by kernel calls on behalf of untrusted user transitions, as illustrated in Figure 26.

The only proof obligation in this case that has to be proved per system and that is not yet discharged once and for all is that the invariant does not depend on changes untrusted components are authorised to perform. If the invariant is written appropriately, this should be a low-effort, potentially automatic proof.

The case where the switch to kernel mode is due to a timer interrupt instead of a kernel call follows precisely the same reasoning and should be even simpler, because in a timer interrupt, the kernel does not perform any authority-relevant actions.

4.7 Trusted component execution up to a kernel call, with interrupts (case 5)

In the last case an interrupt occurs during the execution of a trusted component, say u_1, leading to the execution of another user u_2 being interleaved into the execution of u_1 (as we have seen in Figure 13). As explained in Section 4.3, our approach to reason about trusted components execution

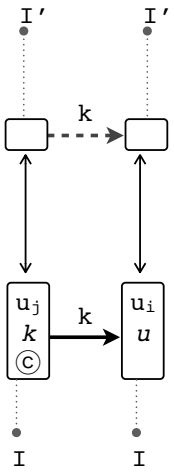


Figure 26: Proof scheme for case 4

was to do the verification of trusted components behaviour at an abstract level, where it requires less effort (as illustrated in Figure 19). The problem is that a sequence of steps in the concrete level corresponds to a single bigger step in the abstract level. Therefore, we can only keep the reasoning at the abstract level if we observe this exact sequence of transitions at the low level. This will not be the case anymore if a timer interrupt occurs during this execution.

The approach here will thus be to prove that we can equivalently *re-order* the execution, so that we keep all u_1 steps in sequence to allow our refinement to hold. Namely, we want to prove that the invariant I holds at each step of the execution if and only if it holds on the execution where the interrupt occurred earlier, as illustrated in Figure 27. The argument is that: (1) timer interrupts do not change any memory content, and (2) u_1 transitions only change u_1 memory, which is disjoint from u_2 memory, because we assume there is no memory sharing between components. Therefore, the u_2 memory is the same in state s_2 as in state s_1 and again the same as in s_0; therefore the behaviour of u_2 should be the same whether it starts executing in s_2 or s_0. Conversely, the execution of u_2 can only affect u_2 private memory and u_1 will therefore find the same state as in its execution before the reordering.

The key observation to note is that the synchronisation points in the system are the kernel calls of components. Re-ordering over timer interrupts is easy, as long as we manage to prove that the kernel protects integrity, authority confinement and confidentiality during this event. Re-ordering over kernel calls of trusted components is not always possible in general—the trusted component may set the system policy into a new state, for instance removing the whole untrusted component u_2 from the system. This fact would clearly be observable and the re-ordered execution would not be equivalent to the original one. We therefore plan to re-order the interleaved execution of u_2 before the execution of u_1. This takes care of u_1 kernel calls.

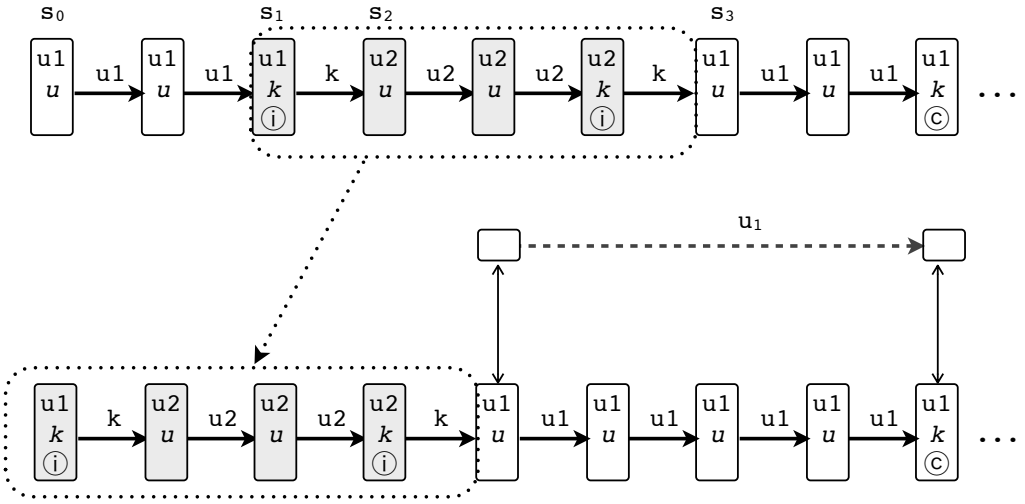


Figure 27: Reordering theorem, for case 5

The most interesting case in the reordering occurs when u_2 performs a kernel call at the end of its execution (instead of another timer interrupt occurring). In this case we need to argue that this kernel call does not affect the private execution of u_1. Again we argue from the policy $\mathcal{P}$ through the integrity theorem and authority confinement that u_2 does not possess enough authority to do anything that affects the private memory or registers of u_1. The kernel call does potentially change kernel state that may affect the next kernel call of the next component, but our re-ordering preserves the order of kernel calls—it just moves u_1 executions over u_2 executions or u_2 kernel calls to the right. By not changing the order of kernel calls our re-ordering preserves the order of synchronisation events in the system and makes sure that each kernel call sees equivalent initial states before and after re-ordering.

The same argument applies if u_2 is itself another trusted component. For this we additionally need the proof obligation that trusted components do not make system policy changes that are observable in the private user-level execution of another trusted component.

Equivalently re-ordering executions this way enables us to deal with the first part of the (re-ordered) execution using case 3 for u_2 (and cases 2 and 4 for the kernel transitions) and deal with the second part for u_1 using case 1.

5 Outcomes of Option 2 Phase

In the last phase of the project, the Option 2 phase, the effort was concentrated on:

- strengthening the isolation guarantees of the kernel by proving the confidentiality property, which combined with integrity and authority confinement reported in Section 3.2, yields the classical isolation property of intransitive noninterference. This activity was of larger scope, but supported in part by this grant.
- formalising the whole system semantics, following the detailed methodology explained in Section 4.

The proof of confidentiality and noninterference is described in [14]; we explain its main points in Section 5.1. The formalised framework prototype for whole system verification has been reported in [4]; we highlight the main results in Section 5.2.

5.1 Noninterference and Confidentiality

5.1.1 Overview

This work involved firstly defining a generic formulation of intransitive noninterference for seL4, which we discuss below in Section 5.1.2. This required defining and proving correct the proof obligations, called *unwinding conditions*, for our noninterference formulation. These two unwinding conditions correspond to integrity and confidentiality, respectively. The integrity unwinding condition is discharged by applying the previous proof of integrity enforcement for seL4 (see Section 3.2).

The bulk of the work of proving noninterference for seL4, then, concentrated on proving that it satisfies confidentiality, in order to discharge the confidentiality unwinding condition. We discuss this further in Section 5.1.3 below. To this end, we defined a generic confidentiality property and an associated proof calculus for it over nondeterministic state monads, which is the formalism in which the seL4 abstract specification is expressed. Our confidentiality property is carefully constructed to be preserved by refinement. This allowed us to prove confidentiality of the abstract specification, and conclude that it holds for the C code that implements seL4 by virtue of the previous functional correctness proofs.

By the end of this phase of the project, we have completed the confidentiality proofs for about 98% of the seL4 abstract specification. This leaves only two kernel functions for which confidentiality is yet to be proved: scheduling and interrupt handling. These functions remain unproved because neither of them, as currently implemented, satisfy confidentiality. For instance, the ordinary seL4 scheduler does not adequately isolate components from each other, because it allows the scheduling decisions about one component to depend on the behaviour of another. This occurs because the scheduler will always schedule a higher priority task in preference to a lower priority one, which allows the lower priority task to infer when the higher priority one is active, even if the two tasks are otherwise unable to communicate with each other. Interrupt handling can

also leak information because seL4 allows one component to be pre-empted while interrupts are delivered to a second component, thereby opening up the possibility for the first component to infer the arrival of interrupts for the second.

A traditional separation kernel avoids each of these information leaks, by constraining the behaviour of the scheduler and enforcing a stricter interrupt delivery regime. We conclude from this work that each of these functions of the seL4 kernel will need to be modified to behave more like a traditional separation kernel in order to complete the proof of confidentiality, and hence noninterference, for seL4.

5.1.2 Noninterference for Operating System Kernels

The isolation property of noninterference divides a system into a set of security *domains*. An *information flow policy* $\rightsquigarrow$ specifies the allowed information flows between domains, such that $d \rightsquigarrow d'$ means that information is allowed to flow from domain d to domain d'. So-called *intransitive noninterference* generalises noninterference to the case in which the information flow policy $\rightsquigarrow$ is possibly intransitive.

For seL4, recall that integrity and authority confinement are defined with respect to an abstract *access control policy* p, represented as a directed graph whose nodes are component labels, representing individual system components, and whose edges carry authority types, representing the authority that components have other each other.

We defined a set of rules for translating an access control policy p to a corresponding information flow policy $\rightsquigarrow$, which has a domain for each of the components (labels) in p as well as an extra domain $\mathsf{s\_dom}$ for the scheduler. Whether information may flow from a domain d to another d' (i.e. whether $d \rightsquigarrow d'$) depends on the authority that d has over d' and vice versa, according to the access control policy p. For instance, $d \rightsquigarrow d'$, if d has Write authority over d' or d' has Read authority over d.

Noninterference asserts that all information flows in the system are in accordance with the information flow policy $\rightsquigarrow$.

Our noninterference formulation assumes the existence of a set of equivalence relations $\overset{d}{\sim}$ on states, one for each domain d, such that $s \overset{d}{\sim} t$ if and only if states s and t agree on the internal state of domain d. Here, d's state will include the user-visible state that d can directly read, but might also include kernel-level state that the kernel might legitimately reveal to d. When the system transitions directly from state s to state s' and $s \overset{d}{\sim} s'$ for example, domain d has not been observably affected by this transition. For a set of domains D, let $s \overset{D}{\approx} t \equiv \forall d \in D. s \overset{d}{\sim} t$.

As is usual, the function Step models individual execution steps of the system, such that $(s, s') \in \mathsf{Step}\ a$ means that the system can transition from state s to state s' when action a occurs.

Traditional noninterference formulations associate a domain $\mathsf{dom}\ a$ with each action a that can occur, such that $\mathsf{dom}\ a$ is the domain that performs the action a. However, for an operating system kernel like seL4, no such intrinsic association exists. Rather, when an action like a system call occurs, the kernel must consult its private data structures to determine which is the currently running

thread, in order to work out which domain has initiated the action a. Therefore, the association from actions a to domains d depends on the current state s of the system.

For this reason, in our noninterference formulation, the domain that performs some action a in a state s is denoted $\mathsf{dom}\ a\ s$. This makes the entire noninterference formulation state-dependent, and somewhat complicates its associated unwinding conditions.

When the scheduler's state is identical in states s and t, we expect that $\mathsf{dom}\ a\ s = \mathsf{dom}\ a\ t$ for all actions a. Recall that the domain $\mathsf{s\_dom}$ encompasses the scheduler state. Therefore, we assume that for all actions a and states s and t:

$$s \overset{\mathsf{s\_dom}}{\sim} t \longrightarrow \mathsf{dom}\ a\ s = \mathsf{dom}\ a\ t$$

Since the scheduler can schedule any domain, and thereby possibly send it information, we assume that the information flow policy $\rightsquigarrow$ allows information to flow from the scheduler domain to all others d:

$$\mathsf{s\_dom} \rightsquigarrow d$$

To prevent the scheduler acting as a global (transitive) channel for information flow, we therefore require that no domain d other than the scheduler can affect (i.e. send information to) $\mathsf{s\_dom}$:

$$d \rightsquigarrow \mathsf{s\_dom} \longrightarrow d = \mathsf{s\_dom}$$

This restriction forces us to prove that the scheduler's decisions about which domain should execute next never depend on the actions or state of any other domain, which is typical scheduler behaviour in a separation kernel. As explained earlier, seL4's current classic priority-base round-robin scheduler does not meet this requirement, which is why we cannot prove that it satisfies confidentiality.

Traditionally, intransitive noninterference definitions make use of a sources function, whereby for a sequence of actions as and a domain d, $\mathsf{sources}\ as\ d$ gives the set of domains that are allowed to pass information to d when as occurs. Because our dom function depends on the current state s, sources must do so as well. Therefore $\mathsf{sources}\ as\ s\ d$ denotes the set of domains that can pass information to d when as occurs, beginning in state s. The following definition is an extension of the standard one [17, 20] in line with our augmented dom function.

$$\begin{array}{rcl}
\mathsf{sources}\ []\ s\ d & = & \{d\} \\
\mathsf{sources}\ (a\#as)\ s\ d & = & (\bigcup\{\mathsf{sources}\ as\ s'\ u \mid s'.\ (s,s') \in \mathsf{Step}\ a\}) \cup \\
 & & \{w.\ w = \mathsf{dom}\ a\ s \wedge \\
 & & \quad (\exists\ v\ s'.\ \mathsf{dom}\ a\ s \rightsquigarrow v \wedge (s,s') \in \mathsf{Step}\ a \wedge v \in \mathsf{sources}\ as\ s'\ u)\}
\end{array}$$

As is usual, the sources function is used to define a purge function, ipurge, in terms of which noninterference is formulated. Naturally, we must break with tradition and include the current state s in our ipurge function. However, for nondeterministic systems purging may proceed from a set ss of possible initial states. This leads to the following definition.

$$\begin{array}{rcl}
\mathsf{ipurge}\ d\ []\ ss & = & [] \\
\mathsf{ipurge}\ d\ (a\#as)\ ss & = & (\mathrm{if}\ (\exists s \in ss.\ \mathsf{dom}\ a\ s \in \mathsf{sources}\ (a\#as)\ s\ d)\ \mathrm{then} \\
 & & \quad a\#\mathsf{ipurge}\ d\ as(\bigcup s \in ss.\ \{s'.\ (s,s') \in \mathsf{Step}\ a\}) \\
 & & \mathrm{else}\ \mathsf{ipurge}\ d\ as\ ss)
\end{array}$$

Given states s and t and action sequences as and bs and a domain d, runequiv $s\ as\ t\ bs\ d$ denotes when executing as from s and bs from t results in states that are identical with respect to domain d, (i.e. that are related under $\overset{d}{\sim}$).

Noninterference is then defined as the predicate noninterference as follows.

$$\begin{aligned}&\mathsf{noninterference} \equiv \forall\ d\ as\ bs\ s\ t.\\ &\quad s \overset{\mathsf{sources}\ as\ s\ d}{\approx} t \wedge s \overset{\mathsf{s\_dom}}{\sim} t \wedge \mathsf{ipurge}\ d\ as\ \{s\} = \mathsf{ipurge}\ d\ bs\ \{s\} \longrightarrow\\ &\quad\ \mathsf{runequiv}\ s\ as\ t\ bs\ d\end{aligned}$$

It says that if two states s and t are equivalent for all domains that pass information to d when the action sequence as executes from s, and are also equivalent with respect to the scheduler domain s_dom, then execution as from s should be indistinguishable to domain d from executing any other action sequence bs from t, provided bs and as agree on the sequence of actions that is allowed to influence d.

This property asserts that neither changes in state, nor actions, outside of the domains allowed to affect d can actually affect domain d.

noninterference talks about entire execution traces as and bs, which makes it difficult to reason about. Fortunately, it is equivalent to the combination of the following *unwinding conditions*, which deal just with individual execution steps. The first, integrity, is an integrity property, while the second, confidentiality, is a confidentiality property.

$$\mathsf{integrity} \equiv \forall\ a\ d\ s\ s'.\ \mathsf{dom}\ a\ s \not\rightsquigarrow d \wedge (s, s') \in \mathsf{Step}\ a \longrightarrow s \overset{d}{\sim} s'$$

$$\begin{aligned}&\mathsf{confidentiality} \equiv \forall\ a\ d\ s\ t\ s'\ t'.\\ &\quad s \overset{\mathsf{s\_dom}}{\sim} t \wedge \mathsf{dom}\ a\ s \rightsquigarrow d \wedge s \overset{\mathsf{dom}\ a\ s}{\sim} t \wedge s \overset{d}{\sim} t \wedge (s, s') \in \mathsf{Step}\ a \wedge (t, t') \in \mathsf{Step}\ a \longrightarrow\\ &\quad\ s' \overset{d}{\sim} t'\end{aligned}$$

Theorem 8. $\mathsf{noninterference} = (\mathsf{integrity} \wedge \mathsf{confidentiality})$

Thus it suffices to prove integrity and confidentiality in order to prove noninterference. noninterference, confidentiality and integrity are all preserved by refinement. This means that if we prove them about the seL4 abstract specification, we can conclude them for its C implementation by Theorem 3.

We apply the seL4 integrity theorem, Theorem 4, in order to prove that integrity holds. To prove confidentiality, we devised a generic confidentiality formulation and proof calculus for nondeterministic state monads, which is the formalism in which the seL4 abstract specification is expressed.

5.1.3 Confidentiality for Nondeterministic State Monads

We formalise confidentiality generically by the property ev, which stands for *equivalence valid*. For equivalence relations A and B, a precondition P and a computation f, ev $A\ B\ P\ f$ asserts

that, for two executions of f each beginning in states that satisfy the precondition P, whenever the initial states are related by A, the final states are related by B and both executions return the same return-value.

For instance, confidentiality amounts to saying $\mathsf{ev}\ \overset{\{\mathsf{s\_dom},\mathsf{dom}\ a\ s,d\}}{\approx}\ \overset{d}{\sim}\ (\lambda s.\mathsf{dom}\ a\ s \rightsquigarrow d)\ f$, for a computation f that implements the Step relation.

In order to decompose proofs of ev across the entire abstract kernel specification, we developed proof rules for each of the primitive monadic operations, $>\!\!>\!\!=$ (pronounced "bind") and return. $>\!\!>\!\!=$ is used to sequence computations together, such that $f >\!\!>\!\!= g$ is the computation that runs f to produce a return-value rv, and then runs $g\ rv$ (i.e. calls g passing it the return-value from f). return x is the computation that simply returns the value x, leaving the system state unmodified.

Our proof calculus for ev builds on top of a Hoare logic for nondeterministic state monads. Here, a Hoare triple is written $\{P\}\ f\ \{Q\}$ and denotes that whenever f is run from a state satisfying the precondition P, then it yields a return-value rv such that the final state satisfies $Q\ rv$ (i.e. Q called with the return-value rv yields the postcondition).

The proof rule for $>\!\!>\!\!=$ is as follows.

$$\frac{\forall rv.\ \mathsf{ev}\ B\ C\ (Q\ rv)\ (g\ rv) \quad \mathsf{ev}\ A\ B\ P'\ f \quad \{P''\}\ f\ \{Q\}}{\mathsf{ev}\ A\ C\ (P'\ \mathsf{and}\ P'')\ (f >\!\!>\!\!= g)}\ \textsc{bind-ev}$$

Here, P' and P'' is the conjunction of preconditions P' and P''. BIND-EV can be read as a recipe for finding a precondition $?P$ such that $\mathsf{ev}\ A\ C\ ?P\ (f >\!\!>\!\!= g)$ is true. First, for any return value rv that f might emit, find some state-equivalence B and a precondition $Q\ rv$, which may mention rv, such that $g\ rv$ yields post-states that satisfy the post-equivalence C. Secondly, find a precondition P' such that executing f yields post-states that satisfy the just found state-equivalence B. Finally, find a precondition P'', such that for all return-values rv emitted from executing f, their corresponding result-states satisfy $Q\ rv$. The desired precondition $?P$ is then P' and P''.

Notice that the post-equivalence, $\overset{d}{\sim}$, for confidentiality is implied by its pre-equivalence, $\overset{\{\mathsf{s\_dom},\mathsf{dom}\ a\ s,d\}}{\approx}$. We may therefore strengthen the post-equivalence to make it identical to the pre-equivalence, thereby allowing us to reason about a property $\mathsf{ev}\ A\ A\ P\ f$ whose pre- and post-equivalences are identical.

This admits a simplified rule for $>\!\!>\!\!=$ in which $B = C = A$, called BIND-EV'.

$$\frac{\forall rv.\ \mathsf{ev}\ A\ A\ (Q\ rv)\ (g\ rv) \quad \mathsf{ev}\ A\ A\ P'\ f \quad \{P''\}\ f\ \{Q\}}{\mathsf{ev}\ A\ A\ (P'\ \mathsf{and}\ P'')\ (f >\!\!>\!\!= g)}\ \textsc{bind-ev'}$$

Applying BIND-EV' requires simply finding appropriate preconditions P' and P''. An ordinary Hoare logic VCG, developed during the functional correctness proofs of seL4, may be used to compute P'', while BIND-EV' is itself a recipe for computing P' and $Q\ rv$. By using the simplified rule BIND-EV', we can therefore perform confidentiality proofs relatively automatically, by recursively applying BIND-EV' and applying the pre-existing Hoare logic VCG.

This requires proof rules for the other primitive monadic operations. The proof rule for return, for instance, follows.

$$\frac{}{\mathsf{ev}\ A\ A\ \top\ (\mathsf{return}\ x)}\ \textsc{return-ev}$$

Here $\top$ is the precondition that holds for all states. RETURN-EV says that any pre-equivalence that holds before return x executes holds afterwards (since return modifies nothing), and two executions of return x return the same value (namely x).

This proof calculus allows fairly automatic proofs for parts of the kernel that never read confidential state. For other parts of the kernel that read (but do not reveal) confidential state, we require a slightly more sophisticated proof calculus that generalises the one above, but admits less automation.

This proof calculus involves the property ev2 that generalises ev. ev2 takes two preconditions P and P' and two computations f and f', as well as a return-value relation R. ev2 $A\ B\ R\ P\ P'\ f\ f'$ says that when executing f from a state satisfying P, and f' from a state satisfying P', then if the pre-states satisfy A, the post states will satisfy B and the two return-values will be related by R. When $P' = P$ and $f' = f$ and R is the equality relation, ev2 $A\ B\ R\ P\ P'\ f\ f' = ev\ A\ B\ P\ f$. Therefore, ev2 generalises ev.

Intermediate computations that read confidential state may yield return values that do not match exactly (if those return values are derived from confidential state). If subsequent computations never affect non-confidential state in a way that depends on these return-values, then the final non-confidential state will not depend on confidential inputs, and so confidentiality will be preserved. However, these subsequent computations may be different, because they take different return-values as inputs. This is why ev2 allows reasoning about two possibly different computations that may return different (but related) return-values.

The proof rule for ev2 over $\gg\!=$ is shown below, and is the obvious generalisation of BIND-EV.

$$\frac{\begin{array}{l}\forall\ rv\ rv'.\ R'\ rv\ rv'\ \longrightarrow \mathsf{ev2}\ B\ C\ R\ (Q\ rv)\ (Q'\ rv')\ (g\ rv)\ (g'\ rv') \\ \mathsf{ev2}\ A\ B\ R'\ P\ P'\ f\ f'\quad \{S\}\ f\ \{Q\}\quad \{S'\}\ f'\ \{Q'\}\end{array}}{\mathsf{ev2}\ A\ C\ R\ (P\ \mathsf{and}\ S)\ (P'\ \mathsf{and}\ S')\ (f \gg\!= g)\ (f' \gg\!= g')}\ \textsc{bind-ev2}$$

The requirement to reason about intermediate return-value relations means that the application of BIND-EV2 cannot be automated in the same way as BIND-EV'. This rule must currently be applied manually, with the human often supplying the intermediate return-value relation.

Nevertheless, the proof calculus for ev2 has allowed us to complete the confidentiality proofs for the major parts of seL4 that read confidential state: namely IPC via asynchronous endpoints.

5.1.4 Conclusion

As mentioned earlier, we have proved confidentiality for about 98% of the seL4 abstract specification, with the scheduler and interrupt handling the two remaining pieces of the puzzle. Neither of these functions currently satisfy confidentiality, because both allow unwanted information leaks.

We conclude from this work that these two functions should be modified to behave more like a traditional separation kernel. Specifically, the scheduler needs to be modified to ensure that its scheduling decisions about one domain never depend on the behaviour of another domain. The interrupt handling routines also need to be modified to ensure that one domain cannot learn about the arrival of interrupts destined for another domain. These modifications will allow the proof of confidentiality to be completed. This will then allow the proof of noninterference to be completed for seL4, which will prove that it enforces strong isolation between distrusting components.

5.2 Formalised framework for whole system verification

The formal proof of noninterference presented in the previous section, together with the functional correctness proof of the seL4 microkernel, form the building blocks to formally prove that a given security or safety property P is true about a given complex system S composed of components running on top of seL4.

The overall aim of the project is to facilitate this kind of formal verification by providing a framework where various intermediate properties about the different parts of a system can be automatically combined, in a provably correct way, into a theorem that P holds at the source code level of the whole system. These intermediate properties, or *proof obligations*, are then given to the user for manual or interactive verification.

Our framework follows the approach described in Section 4 and an initial prototype has been formalised in Isabelle/HOL, as reported in [4]. We highlight here the main characteristics of this formalisation.

Since the aim is to minimize the verification effort for the user, we need to minimise the set of proof obligations as much as possible (i.e. weaken the properties to be proved by the user), by following a defined proof strategy, and discharging (i.e. automatically proving) any intermediate lemma needed to weaken the properties. In particular, our framework assumes the system follows a strategy of verified, minimal computing base, and therefore requires as input a proof of the kernel's correctness and isolation properties, and a proof that the trusted components satisfy the targeted property. Moreover, as already mentioned, we will focus in this project on properties that can be derived from a system invariant I, i.e. we constrain the problem space to the class of formal safety properties. In practice, this means that the goal comes down to proving that

I is preserved by any execution step t of the system.

where the system is represented as a state machine with transitions t between system states. For now, we treat the (usually more difficult) step case during system execution. For the property P to hold over the entire system execution, we must additionally show that the initial state of the system also satisfies the invariant I. This would link to our work on correct initialiser reported in AOARD project #FA2386-11-1-4070 [5].

The framework is generic in that it refers to any property that can be derived for a system invariant and to any kernel which can provide isolation and functional correctness guarantees.

The decomposition of the final invariant preservation goal into several proof obligations will follow the strategy explained in Section 4, namely using two main proof techniques:

- ***Abstraction***: here we first show that a specific kind of concrete transition tc can be abstracted to a simpler (abstract) transition ta where it is easier to reason about the invariant preservation; this will be done using refinement techniques, namely forward simulation, illustrated in Figure 28. The invariant preservation is then shown at the abstract level. This means that each

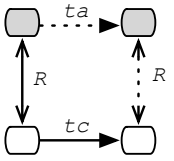


Figure 28: Forward simulation between concrete transition tc and abstract transition ta.

  transition for which this technique is used can be proved at a different level of abstraction, enabling us to choose the level best suited to each particular problem.

- ***Locality of state changes***: here we use the fact that parts of the state may stay unchanged during given transitions, and if the invariant I does not depend on what might have changed, then I is trivially preserved. We will note $s =_{|M} s'$ when the state s' is equal to the state s up to a given substate M. If I does not depend on M and $s =_{|M} s'$ where s (resp. s') is the state before (resp. after) a transition, then this transition preserves I. We use this when reasoning about transitions on behalf of untrusted users: if isolation is enforced by the kernel, such transitions may only change what the untrusted components may modify. Now, by definition of untrusted components, the invariant I should not depend on the parts of the state that these components have access to. This leads us to place a reasonable assumption on the invariant to be discharged by the user of the framework.

Using these two techniques, we can decompose the proof that invariant I is preserved by transition t (where each step of t is either a kernel transition tk or a user transition tu or the identity over states) into proofs at a different level of abstraction or proof obligations that I is independent of some substate.

As explained in detail in [4], this gradual decomposition and weakening of the proof obligations is formalised using Isabelle's *locale* mechanism. A locale l_1 contains a set of definitions, assumptions and theorems, and we can declare another locale l_2 as being a *sublocale* of l_1. This requires us to prove (once and for all) that all the assumptions (in our case proof obligations) of l_1 can be derived from l_2 ones. This in turn enables us to replace a strong proof obligation in l_1 by a weaker or a set of smaller proof obligations in l_2. What is automatically granted in the process is that all theorems proved in the context of l_1 are automatically inherited in l_2, because they were proved using l_1 assumptions, which can be derived from l_2 ones.

The formal framework is thus defined as a hierarchy of sublocales, whose root contains the overall theorem stating the invariant preservation for the global system transition, and the "last" sublocale contains proof obligations that are as weak as possible, as illustrated in Figure 29.

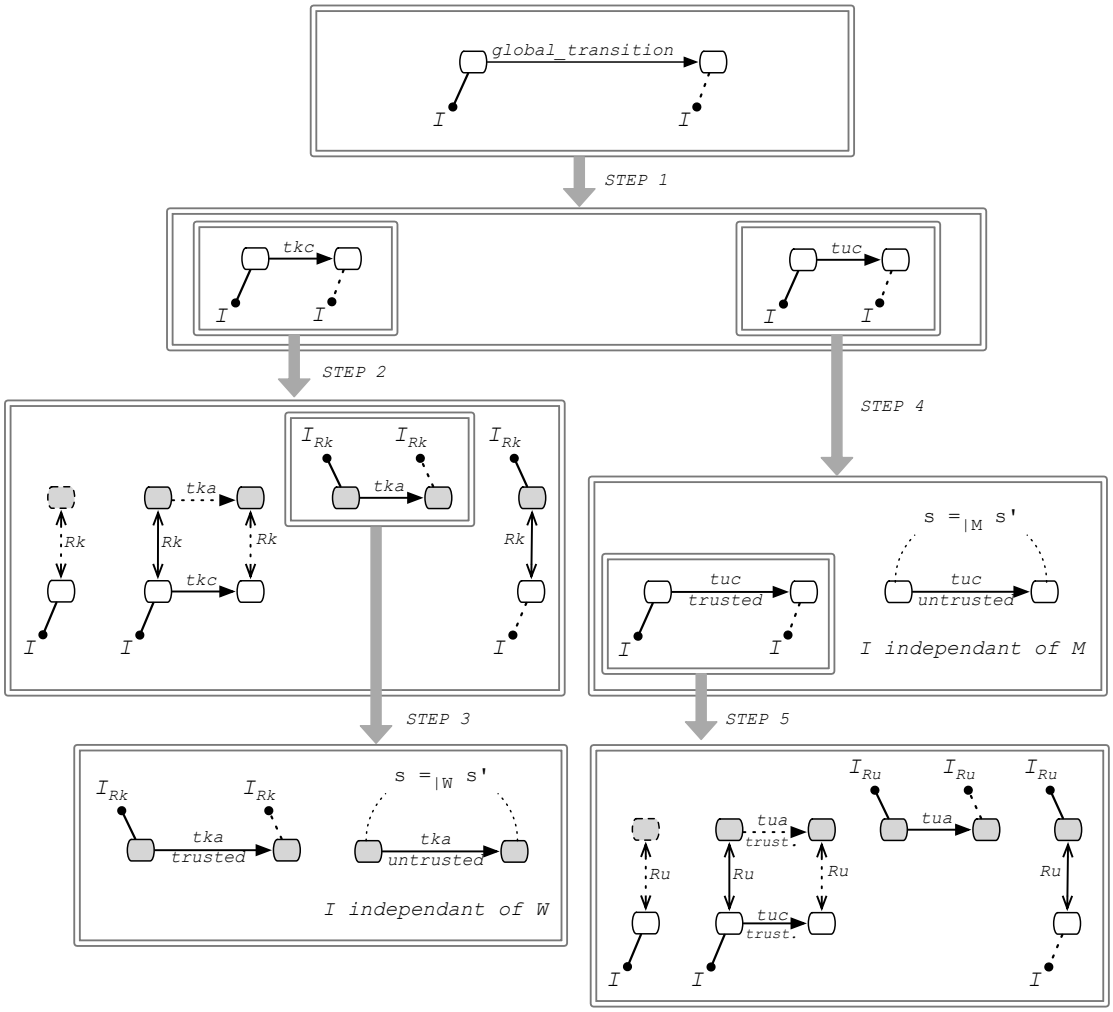


Figure 29: Structure of the locales hierarchy and proof overview.

The first step consists in doing a case distinction on the kind of transition, i.e. kernel transition or user transition. In the second step, we push the invariant preservation proof for kernel transition at a more abstract level, assuming a forward simulation proof for kernel transition (and two necessary conditions on the state relation). In the third step, we decompose the invariant preservation for abstract kernel transition into two cases, depending on whether the running component is trusted or untrusted; and in the untrusted case, we can replace the invariant preservation by a proof of integrity

preservation and a proof that the invariant in independent of what untrusted components can modify. In a fourth step, we look at user transition and do a case distinction again on whether the running component is trusted or untrusted; and in the untrusted case, we replace the invariant preservation by a proof that I is independent of untrusted memory content and a proof that user transitions only change user memory. In a last step, we speculate that we can get a refinement proof for trusted component behaviour, pushing the invariant preservation proof of trusted user to a more abstract level.

To summarise, in the end the framework contains three types of proof obligations. Firstly, we have three proof obligations about the kernel or system automaton, that do not depend on the specific system built on top:

- proof of functional correctness for the kernel;
- proof that the kernel enforces integrity;
- proof that user transitions may only change user memory.

These three proof obligations can be discharged once for a given kernel and the immediate next step of our project is to partially instantiate the framework with the seL4 microkernel and discharge those assumptions.

The next kind of proof obligations are assumptions about the invariant that should be relatively easy to discharge given the definition of untrusted components and targeted property:

- proof that the invariant is independent of what untrusted users can modify and their memory (i.e. that untrusted users really need not be trusted);
- proof that the invariant satisfies the totality conditions about the state relations used in the kernel's and the trusted user's functional correctness proof. These are necessary to lift the invariant preservation to an abstract level.

Finally, the last category of proof obligations are the more involved proofs about the behaviour of trusted components. By definition, trusted components can potentially violate the targeted property and hence the system invariant. We therefore need a formal model of their behaviour and a proof that they preserve the invariant, both during purely user-level transitions and during kernel calls that they initiate:

- proof of functional correctness for trusted, concrete, small-step, user transitions;
- proof that trusted kernel calls preserves the invariant

These two last assumptions are more speculative and lead to further work in improving or validating the framework.

6 Conclusion and Future Directions

This report has summarised the work of AOARD project #FA2386-10-1-4105 including the content of its related annual reports as well as the final reports for intermediate option phases. We have also included work funded in part by the extension grant #FA2386-12-1-4022 [4].

The goal of this AOARD project #FA2386-10-1-4105 was to investigate the creation of a formal overall system framework that enables the use of kernel correctness, component correctness and security analysis to conclude one formal, machine-checked theorem about the overall system on the level of its implementation. A specific objective was to investigate how to integrate a number of heterogeneous formal analyses into a single framework such that the overall effort for verification of the whole system is reduced to a provably correct combination of automatic, manual, and semi automatic partial proofs.

These goals were both achieved. We have created and demonstrated an initial framework, formalised in the theorem prover Isabelle/HOL, in which such proofs can be conducted.

In particular, the work under the grant lead to the following key research outcomes.

- The formal whole-system framework mentioned above, which enables reasoning about MILS-style systems at different levels of abstraction.
- The detailed formalisation of the behaviour of a trusted component, which, combined with a previous higher-level formalisation, demonstrates the use of different levels of abstraction from code reasoning up to model checking.
- An integrity proof for the seL4 microkernel the first such proof for any general-purpose OS on the code-level. This satisfies one key assumption of the above framework necessary to reduce effort in formal reasoning about systems.
- A formal calculus for proving confidentiality of microkernels, applied to 98% to the seL4 microkernel. When completed, this work will satisfy a second key assumption (next to functional correctness) of the above framework.

Both the integrity and confidentiality results where larger-scale activities than this project, but they were in part supported by this grant. The final piece of the puzzle will be provided by extension grant #FA2386-11-1-4070 on system initialisation, which has already produced first results [5].

Future work. While the work under grant #FA2386-10-1-4105 has made fundamental advances in reducing the effort for reasoning about the security and dependability of MILS-style high-assurance systems, important engineering work is still necessary to use it in practise. To complete the confidentiality proof for instance, the seL4 kernel will need to be modified to support a separation-style scheduler. To apply the formal whole-system framework to more realistic systems, it needs to be extended to include interrupts, devices, and better support for reasoning about the interleaved or concurrent execution of multiple components, for instance by assume-guarantee reasoning.

References

[1] Jim Alves-Foss, Paul W. Oman, Carol Taylor, and Scott Harrison. The MILS architecture for high-assurance embedded systems. *International Journal on Embedded Systems*, 2:239–247, 2006.

[2] 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.

[3] June Andronick and Gerwin Klein. Formal system verification for trustworthy embedded systems, final report aoard 094160. http://www.nicta.com.au/pub?doc=5614, NICTA, Sydney, Australia, April 2011.

[4] June Andronick and Gerwin Klein. Formal system verification - extension 2, final report AOARD FA2386-12-1-4022. http://www.nicta.com.au/pub?doc=6302, NICTA, Sydney, Australia, August 2012.

[5] June Andronick, Gerwin Klein, and Andrew Boyton. Formal system verification — extension, annual report AOARD 114070. http://www.nicta.com.au/pub?doc=5926 ISSN 1833-9646-5926, NICTA, Sydney, Australia, May 2012.

[6] June Andronick, Gerwin Klein, and Toby Murray. Final report option 1 AOARD 104105, formal system verification for trustworthy embedded systems. http://www.nicta.com.au/pub?doc=5617 ISSN 1833-9646-5617, NICTA, Sydney, Australia, November 2011.

[7] D.E. Bell and L.J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report MTR-2997, MITRE Corp., March 1976.

[8] Nelson Billing. Formal functional specification of a security critical component in Isabelle/HOL. Bachelor thesis, University of NSW, School of Computer Science and Engineering, April 2012.

[9] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending sledgehammer with SMT solvers. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *CADE*, volume 6803 of *Lecture Notes in Computer Science*, pages 116–130. Springer, 2011.

[10] Sascha Böhme and Tobias Nipkow. Sledgehammer: Judgement day. In Jürgen Giesl and Reiner Hähnle, editors, *IJCAR*, volume 6173 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2010.

[11] 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 operating system kernel. *Communications of the ACM*, 53(6):107–115, June 2010.

[12] Butler W. Lampson. Protection. In *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems*, pages 437–443, Princeton University, March 1971. Reprinted in ACM Operating Systems Review, **8**(1), export-jan 1974, pp 18–24.

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

[14] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. Noninterference for operating system kernels. In *Proceedings of the 2nd International Conference on Certified Programs and Proofs*, December 2012. To appear.

[15] 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-Verlag, 2002.

[16] Michael Norrish. C-to-Isabelle parser, version 0.7.2. http://ertos.nicta.com.au/software/c-parser/, Jan 2012.

[17] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, SRI International, December 1992.

[18] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *2nd International Conference on Interactive Theorem Proving*, volume 6898 of *Lecture Notes in Computer Science*, pages 325–340, Nijmegen, The Netherlands, August 2011. Springer-Verlag.

[19] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *34th POPL*, pages 97–108. ACM, 2007.

[20] David von Oheimb. Information flow control revisited: Noninfluence = noninterference + nonleakage. In Pierangela Samarati, Peter Ryan, Dieter Gollmann, and Refik Molva, editors, *Proceedings of the 9th European Symposium On Research in Computer Security*, volume 3193 of *Lecture Notes in Computer Science*, pages 225–243, 2004.

7 Attachments

- Conference paper
  seL4 Enforces Integrity. [18]
- BE thesis
  Formal Functional Specification of a Security Critical Component in Isabelle/HOL. [8]
- Conference paper (accepted for publication)
  Noninterference for Operating System Kernels. [14]

seL4 Enforces Integrity

Thomas Sewell[1], Simon Winwood[1,2], Peter Gammie[1], Toby Murray[1,2], June Andronick[1,2], and Gerwin Klein[1,2]

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

{first-name.last-name}@nicta.com.au

Abstract. We prove that the seL4 microkernel enforces two high-level access control properties: integrity and authority confinement. Integrity provides an upper bound on write operations. Authority confinement provides an upper bound on how authority may change. Apart from being a desirable security property in its own right, integrity can be used as a general framing property for the verification of user-level system composition. The proof is machine checked in Isabelle/HOL and the results hold via refinement for the C implementation of the kernel.

1 Introduction

Enforcing access control is one of the primary security functions of an operating system (OS) kernel. Access control is usually defined as two properties: confidentiality, which means that information is not acquired without read authority, and integrity, which means that information is not modified without write authority. These properties have been well studied in relation to classical security designs such as the Bell-LaPadula model [3]. For dynamic access control systems, such as the capability system in the seL4 microkernel, an additional property is of interest: authority confinement, which means that authority may not be spread from one subject to another without explicit transfer authority.

We have previously verified the functional correctness of seL4 [12]. In this work we prove that seL4 correctly enforces two high level security properties: integrity and authority confinement.

We define these properties with reference to a user-supplied security policy. This policy specifies the maximum authority a system component may have. Integrity limits state mutations to those which the policy permits the subject components to perform. Authority confinement limits authority changes to those where components gain no more authority than the policy permits. The policy provides mandatory access control bounds; within these bounds access control is discretionary.

While integrity is an important security property on its own, it is of special interest in formal system verification. It provides a framing condition for the

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

execution of user-level components, telling us which parts of the system do not change. In a rely-guarantee framework, the integrity property gives us useful guarantee conditions for components without the need to consult their code. This is because the kernel, not the component, is providing the guarantee. This becomes especially important if the system contains components that are otherwise beyond our means for formal code-level verification, such as a Linux guest operating system with millions of lines of code. We can now safely and formally compose such parts with the rest of the system.

Access control properties and framing conditions have been extensively studied. Proving these properties about a real OS kernel implementation, however, has not been achieved before [10]. Specifically, the novelty and contributions of this work are:

- The Isabelle/HOL formalisation and generalisation of integrity and authority confinement for a real microkernel implementation.
- To the best of our knowledge the first code-level proof of high-level access control properties of a high-performance OS kernel.

Our proof is connected to reality via refinement to the C implementation. This means we must deal with all the complexities and corner cases of the kernel we have, rather than laying out a kernel design which fits neatly with our desired access control model and hoping to implement it later.

We make one kind of simplifying assumption: we place restrictions on the policy. These forbid some kinds of interaction between components which are difficult for us to reason about. Although we have not yet applied the theorem to a large system, our choice of assumptions has been guided by a previous case study [2] on a secure network access device (SAC), with a dynamic and realistic security architecture.

We are confident that a significant variety of security designs will, after some cosmetic adjustments, comply with our restrictions. We support fine grained components, communication between them via memory sharing and message passing, delegation of authority to subsystems and dynamic creation and deletion of objects. We support but restrict propagation of authority and policy reconfiguration at runtime.

In the following, Sect. 2 gives a brief introduction to access control in general and to the seL4 access control system in particular. Sect. 2 also introduces a part of the aforementioned SAC system as a running example. Sect. 3 gives a summary of the formalisation as well as the final Isabelle/HOL theorems and Sect. 4 discusses the results together with our experience in proving them.

2 Access Control Enforcement and seL4

This section introduces relevant concepts from the theory of access control and our approach to instantiating them for seL4. For ease of explanation we will first introduce a running example, then use it to describe the seL4 security mechanisms available, and then compare to the theory of access control.

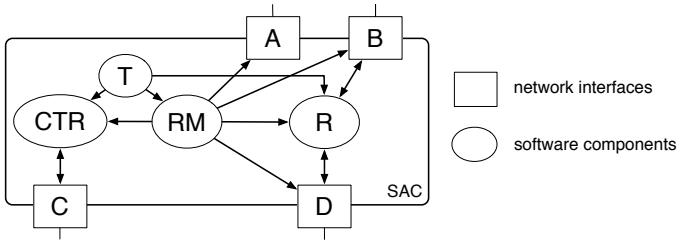


Fig. 1. System snapshot, routing between data D and back-end B network.

2.1 Example

The secure access controller (SAC) was designed in an earlier case study of ours [2] and will serve as a running example in this work. The purpose of the SAC is to switch one front-end terminal between different back-end networks of separate classification levels. The security goal of the system is to avoid information flow between the back-end networks.

Fig. 1 shows the main components of the SAC as nodes and their logical authority connections as edges. To the outside, the SAC provides four network interfaces: two back-end networks A and B, one control network C, and one data network D. Networks C and D are attached to a front-end terminal. The purpose of the SAC is to connect either A to D or B to D at a time without information flow between A and B. Internally, we have four components: a timer T, a controller user interface CTR, the router manager RM, and a router instance R. The router manager RM will upon a switch request, tear down R, remove all access from it, create and start a fresh R component, and connect it to the requested networks. RM is a small, trusted component and has access to all networks. Verification needs to show that it does not abuse this access. The router R, on the other hand, is a large, untrusted instance of Linux. It will only ever be given access to one of the back-end networks during its lifetime.

In previous work [2], we have shown that, assuming a specific policy setup, assuming correct behaviour of RM, and assuming that the kernel correctly enforces access control, the security goal of the system will be enforced.

The work in this paper helps us to discharge the latter two assumptions: we can use the integrity property as a framing condition for verifying the behaviour of RM, and we can use the same property to make sure that R stays within its information flow bounds. To complete the verification, we would additionally need the confidentially side of access control as well as a certified initial policy set up. Both are left for future work.

2.2 seL4

The seL4 microkernel is a small operating system kernel. As a microkernel, it provides a minimal number of services to applications: interprocess communication, threads, virtual memory, access control, and interrupt control.

As mentioned, seL4 implements a capability-based access control system [6]. Services, provided by a set of methods on kernel implemented objects, are invoked by presenting to the kernel a capability that refers to the object in question and has sufficient access rights for the requested method. For the purposes of this paper, the following four object classes are the most relevant.

CNodes Capabilities are stored in kernel-protected objects called *CNodes*. These CNodes can be composed into a *CSpace*, a set of linked CNodes, that defines the set of capabilities possessed by a single thread. CNode methods allow copying, insertion and removal of capabilities. For a thread to use a capability, this capability must be stored in the thread's CSpace. CNodes can be shared across CSpaces. The links in Fig. 1 mean that the collective CSpaces of a component provide enough capabilities to access or communicate with another component.

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. On ARM and Intel IA32 architectures, 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. A Frame can appear in multiple VSpaces, and thereby implement shared memory between threads or devices such as the SAC networks.

Threads Threads are the unit of execution in seL4, the *subjects* in access control terminology. Each thread has a corresponding *TCB* (thread control block), a kernel object that holds its data and provides the access point for controlling it. A TCB contains capabilities for the thread's CSpace and VSpace roots. Multiple threads can share the same CSpace and VSpace or parts thereof. A component in the SAC example may consist of one or more threads.

Inter-process Communication (IPC) Message passing between threads is facilitated by *Endpoints* (EP). The kernel provides *Synchronous Endpoints* with rendezvous-style communication, and *Asynchronous Endpoints* (AEP) for notification messages. Synchronous endpoints can also be used to transfer capabilities if the sender's capability to the endpoint has the *Grant* right. The edges in the SAC example of Fig. 1 that are physical communication links, are implemented using endpoints.

The mechanisms summarised above are flexible, but low-level, as customary in microkernels. Fig. 2 shows parts of the implementation of the link between the CTR and RM components of Fig. 1. The link is implemented via a synchronous endpoint EP. The CTR and RM components both consist of one main thread, each with a CSpace containing a capability to the endpoint (among others), and each with a VSpace containing page directories (pd), page tables (pt), and frames f_n implementing private memory.

These mechanisms present a difficulty for access control due to their fine grained nature. Once larger components are running, there can easily exist hundreds of thousands of capabilities in the system. In addition, seL4 for ARM has

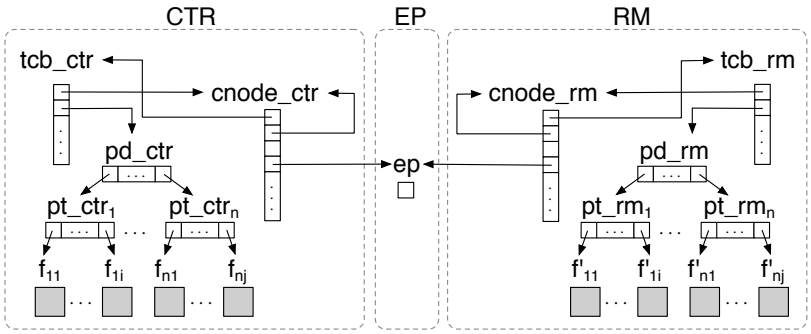


Fig. 2. SAC capabilities (partial).

15 different capability types, several of which have specific rights and variations. We will prune this complexity down by making some simplifying observations. One such observation is that many kinds of capabilities will not be shared without the sharing components trusting each other. For example sharing part of a CSpace with an untrusted partner makes little sense, as seL4 does not provide a mechanism for safely using a capability of unknown type.

2.3 Access Control Enforcement

An access control system controls the access of *subjects* to *objects* [14], by restricting the operations that subjects may perform on objects in each state of the system. As mentioned above, in seL4 the subjects are threads, the objects are all kernel objects, including memory pages and threads themselves.

The part of the system state used to make access control decisions, i.e., the part that is examined by the kernel to decide which methods each subject may perform on which object, is called the *protection state*. In seL4, this protection state is mostly represented explicitly in the capabilities present in the CSpace of each subject. This explicit, fine-grained representation is one of the features of capability-based access control mechanisms. In reality, however, some implicit protection state remains, for instance encoded in the control state of a thread, or in the presence of virtual memory mappings in a VSpace.

The protection state governs not only what operations are allowed to be performed, but also how each subject may modify the protection state. For instance, the authority for capabilities to be transmitted and shared between subjects is itself provided by CNode or endpoint capabilities.

As seen previously in Fig. 2, the protection state of a real microkernel can be very detailed, and therefore cumbersome to describe formally. It is even more cumbersome to describe precisely what the allowed effects of each operation are at this level. Hence we make use of the traditional concept of a policy which can be seen as an abstraction of the protection state: we assign a label to each object and subject, and we specify the authority between labels as a directed graph. The abstraction also maps the many kinds of access rights in seL4 protection states

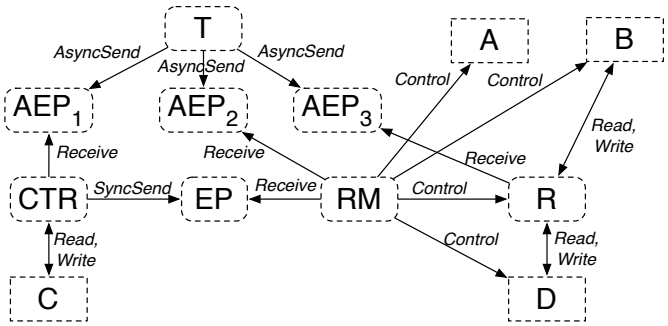


Fig. 3. SAC authority (except self-authority).

into a simple enumerated set of authority types. This simplifies the setup in three ways: the number of labels can be much smaller than the number of objects in the system, the policy is static over each system call whereas the protection state may change and, finally, we can formulate which state mutations are allowed by consulting the policy, rather than the more complex protection state.

The abstraction level of the policy is constrained only by the wellformedness assumptions we make in Sect. 3. Within these bounds it can be chosen freely and suitably for any given system.

Fig. 3 shows an abstract policy, only mildly simplified for presentation, that corresponds to a possible protection state of the SAC at runtime. The objects in the system are grouped by labels according to the intention of the component architecture. The RM label, for instance, includes all objects in the RM box of Fig. 2. The communication endpoints between components have their own label to make the direction of information flow explicit. The edges in the figure are annotated with authority types described in Sect. 3.

Correct access control enforcement, with respect to a security policy, can be decomposed into three properties about the kernel: *integrity*, *confidentiality* and *authority confinement*. The two former properties are the same as introduced in Sect. 1, only the notion of what is permitted is taken from the policy rather than the protection state. The latter property ensures the current subject cannot escalate its authority (or another subject's) above what the policy allows.

Note that we may have some components in the system, such as RM in the SAC, which have sufficient authority to break authority confinement. The authority confinement theorem will assume these components are not the current subject, and we will be obliged to provide some other validation of their actions.

3 Formalisation of Integrity Preservation

This section sketches our Isabelle/HOL formalisation of the integrity and authority confinement properties. While it is impossible in the space constraints of a paper to give the full detail, we show the major definitions and the top level theorems

to provide a flavour of the formalisation. For formal details on the kernel-level concepts beyond our description below we refer the interested reader to the published Isabelle/HOL specification of the kernel [1] that the definitions here build on.

We have already introduced the notion of a policy, an upper bound on the protection state of the system, and an accompanying abstraction, a mapping from detailed subject and object names up to a smaller set of component labels. We roll these objects together with a subject label into the PAS record (policy, abstraction, subject) which is an input to all of our access control predicates.

$$\begin{aligned}\textbf{record}\ 'l\ \mathsf{PAS}\ = \quad & \mathsf{pasPolicy} :: ('l \times \mathsf{auth} \times 'l)\ \mathsf{set} \\ & \mathsf{pasObjectAbs} :: \mathsf{obj\text{-}ref} \Rightarrow 'l \\ & \mathsf{pasIRQAbs} :: \mathsf{irq} \Rightarrow 'l \\ & \mathsf{pasASIDAbs} :: \mathsf{asid} \Rightarrow 'l \\ & \mathsf{pasSubject} :: 'l\end{aligned}$$

The type parameter $'l$ here is any convenient type for component labels. The policy field is a graph (a set of triples $'l \times \mathsf{auth} \times 'l$), whose vertices are policy labels and whose edges are authority types from the type auth which will be discussed shortly. Abstraction functions are provided for seL4's namespaces: objects (i.e. system memory), interrupt request numbers and address space identifiers. Each of these is mapped up to a policy label.

The subject in the PAS record identifies the label of the current subject. We must associate all (write) actions with a subject in order to define the integrity property we are proving. We will pick the subject associated with any kernel actions at kernel entry time, choosing the label of the currently running thread. This coarse division of actions between subjects causes some problems for message transfers, as will be discussed below.

The identification of a subject makes all of our access control work *subjective*. The integrity property depends on which subject carries out an action, because whether that action is allowed or not depends on the allowed authority of the subject performing it. Wellformedness of PAS records, encapsulating our policy assumptions, will be defined in a subjective manner as well, so for any given system and policy the authority confinement proof may be valid only for less-trusted subjects which satisfy our policy assumptions.

3.1 Authority Types

We made the observation in Sect. 2 that most kinds of objects in seL4 are not shared by mutually distrusting components. We found that the objects that could be safely shared were those where authority to that object could be partitioned between capabilities. Endpoints are a good example: Capabilities to endpoints can be send-only or receive-only. Every endpoint in the SAC has a single sending component and a single receiving component, as is typical in seL4 system architectures. Memory frames may also be read only or read-write and a typical sharing arrangement has a single writer, though possibly many readers.

This led us to our chosen formalisation of authority types.

$$\textbf{datatype}\ \mathsf{auth} = \mathsf{Receive} \mid \mathsf{SyncSend} \mid \mathsf{AsyncSend} \mid \mathsf{Reset} \mid \mathsf{Grant} \mid \mathsf{Write} \mid \mathsf{Read} \mid \mathsf{Control}$$

The Receive, Read and Write authorities have been described above. We distinguish endpoint types via SyncSend and AsyncSend. This distinction anticipates future work on confidentiality, where synchronous sends spread information in both directions. Capabilities to endpoints may also have the Grant right which permits other capabilities to be sent along with messages. The Reset authority is conferred by all capabilities to endpoints, since these capabilities can sometimes cause an endpoint reset even if they have zero rights. Finally, the Control authority is used to represent complete control over the target; it is a conservative approximation used for every other kind of authority in the system.

3.2 Subjective Policy Wellformedness

We aim to show authority is confined by a policy. This will not be true for all kinds of policies. If a policy gives the subject a Grant authority to any other component in addition to some authority which the other component should not have, that policy can clearly be invalidated. We define wellformedness criteria on policies, and thereby system architectures, given a specific subject, as follows. In our example Fig. 3, we would expect the policy to be wellformed for the subjects CTR, T, and R, but not RM.

$$\begin{aligned}
&\mathsf{policy\text{-}wellformed}\ \mathit{policy}\ \mathit{irqs}\ \mathit{subject} \equiv \\
&\quad (\forall\, a.\ (\mathit{subject},\ \mathsf{Control},\ a) \in \mathit{policy} \longrightarrow \mathit{subject} = a) \wedge \\
&\quad (\forall\, a.\ (\mathit{subject},\ a,\ \mathit{subject}) \in \mathit{policy}) \wedge \\
&\quad (\forall\, s\ r\ \mathit{ep}. \\
&\qquad (s,\ \mathsf{Grant},\ \mathit{ep}) \in \mathit{policy} \wedge (r,\ \mathsf{Receive},\ \mathit{ep}) \in \mathit{policy} \longrightarrow \\
&\qquad (s,\ \mathsf{Control},\ r) \in \mathit{policy} \wedge (r,\ \mathsf{Control},\ s) \in \mathit{policy}) \wedge \\
&\quad (\forall\, i{\in}\mathit{irqs}.\ \forall\, p.\ (i,\ \mathsf{AsyncSend},\ p) \in \mathit{policy} \longrightarrow (\mathit{subject},\ \mathsf{AsyncSend},\ p) \in \mathit{policy})
\end{aligned}$$

The first requirement is that the subject cannot have Control authority over another component. If it did there would be no point in separating these components, as the subject might coerce the target into taking actions on its behalf.

The second requirement is that the subject has all kinds of authority to itself. We always consider components permitted to reconfigure themselves arbitrarily.

The Grant restriction observes that successful capability transfers over messages are as problematic for access control as Control authority. In each direction this restriction could be lifted if we introduced more complexity.

In the sending direction the problem is that the sender can transfer an arbitrary capability into the receiver's capability space, giving the sender the new authority to rescind capabilities from the receiver's capability space in the future. It may be possible in seL4 for a receiver to partition its capability space to make this safe, but we know of no use case that justifies the resulting complexity.

In the receiving direction the problem is in the way we fix the subject of the message send. Synchronous sends in seL4 complete when both sender and receiver are ready. If the sender is ready when our subject makes a receive system call,

it may appear that the receiver has broken authority confinement by magically acquiring new authority. In fact the authority belonged to the sender, which was involved as a subject in some sense, but not in a manner that is easy to capture.

The final policy assumption relates to interrupts. Interrupts may arrive at any time, delivering an asynchronous message to a thread waiting for that interrupt. We must allow the current subject to send this message. We hope to revisit our simple notion of the current subject in future work.

3.3 Policy/Abstraction Refinement

We define a kernel state s as being a refinement of the policy p as follows.

$$\begin{aligned}&\mathsf{pas\text{-}refined}\ p\ s \equiv \\ &\quad \mathsf{policy\text{-}wellformed}\ (\mathsf{pasPolicy}\ p)\ (\mathsf{range}\ (\mathsf{pasIRQAbs}\ p))\ (\mathsf{pasSubject}\ p) \wedge \\ &\quad \mathsf{irq\text{-}map\text{-}wellformed}\ p\ s \wedge \\ &\quad \mathsf{auth\text{-}graph\text{-}map}\ (\mathsf{pasObjectAbs}\ p)\ (\mathsf{state\text{-}objs\text{-}to\text{-}policy}\ s) \subseteq \mathsf{pasPolicy}\ p \wedge \\ &\quad \mathsf{state\text{-}asids\text{-}to\text{-}policy}\ p\ s \subseteq \mathsf{pasPolicy}\ p \wedge \\ &\quad \mathsf{state\text{-}irqs\text{-}to\text{-}policy}\ p\ s \subseteq \mathsf{pasPolicy}\ p\end{aligned}$$

The kernel state refines the policy if the various forms of authority contained within it, when labelled by the abstraction functions, are a subset of the policy. The full definitions of the extraction functions for authority from the kernel state are too detailed to describe here. In summary, a subject has authority over an object for one of these reasons:

- it possesses a capability to the object.
- it is a thread which is waiting to conclude a message send. For performance reasons the capability needed to start the send is not rechecked on completion, and thus the thread state is an authority in its own right.
- it possesses the parent capability in the capability derivation tree (cdt) of a capability stored in the object.
- the page tables link the subject to the object.
- the active virtual address space database names the object as the page directory for an address space identifier the subject owns.
- the interrupt despatch mechanism lists the object as the receiver for an interrupt request number the subject owns.

Note that none of this authority extraction is subjective. The pas-refined predicate is subjective only because it also asserts policy-wellformed. The reason for this is convenience: these properties are almost always needed together in the proof.

3.4 Specification of Integrity

We define access control integrity subjectively as follows:

$$\begin{aligned}&\mathsf{integrity}\ p\ s\ s' \equiv \\ &\quad (\forall x.\ \mathsf{object\text{-}integrity}\ p\ (\mathsf{pasObjectAbs}\ p\ x)\ (\mathsf{kheap}\ s\ x)\ (\mathsf{kheap}\ s'\ x)) \wedge \\ &\quad (\forall x.\ \mathsf{memory\text{-}integrity}\ p\ x\ (\mathsf{tcb\text{-}states\text{-}of\text{-}state}\ s)\ (\mathsf{tcb\text{-}states\text{-}of\text{-}state}\ s') \\ &\qquad (\mathsf{auth\text{-}ipc\text{-}buffers}\ s)\ (\mathsf{memory\text{-}of}\ s\ x)\ (\mathsf{memory\text{-}of}\ s'\ x)) \wedge \\ &\quad (\forall x.\ \mathsf{cdt\text{-}integrity}\ p\ x\ (\mathsf{cdt}\ s\ x, \mathsf{is\text{-}original\text{-}cap}\ s\ x)\ (\mathsf{cdt}\ s'\ x, \mathsf{is\text{-}original\text{-}cap}\ s'\ x))\end{aligned}$$

This says that a transition from s to s' satisfies access control integrity if all kernel objects in the kernel object heap (kheap s), user memory and the capability derivation tree (cdt s) were changed in an acceptable manner.

The object level integrity predicate is defined by eight introduction rules. The following three rules are representative:

$$\frac{x = \mathsf{pasSubject}\ p}{\mathsf{object\text{-}integrity}\ p\ x\ ko\ ko'} \qquad \frac{ko = ko'}{\mathsf{object\text{-}integrity}\ p\ x\ ko\ ko'}$$

$$\frac{\begin{array}{c} ko = \mathsf{Some}\ (\mathsf{TCB}\ tcb) \qquad ko' = \mathsf{Some}\ (\mathsf{TCB}\ tcb') \\ \exists\, ctxt'.\ tcb' = tcb(\!|\mathsf{tcb\text{-}context} := ctxt', \mathsf{tcb\text{-}state} := \mathsf{Running}|\!) \\ \mathsf{receive\text{-}blocked\text{-}on}\ ep\ (\mathsf{tcb\text{-}state}\ tcb) \qquad \mathsf{auth} \in \{\mathsf{SyncSend}, \mathsf{AsyncSend}\} \\ (\mathsf{pasSubject}\ p, \mathsf{auth}, \mathsf{pasObjectAbs}\ p\ ep) \in \mathsf{pasPolicy}\ p \end{array}}{\mathsf{object\text{-}integrity}\ p\ l'\ ko\ ko'}$$

These cases allow the subject to make any change to itself, to leave anything unchanged, and to send a message through an endpoint it has send access to and into the registers (called the tcb-context here) of a waiting receiver. Note that it is guaranteed that the receiver's registers are changed only if the message transfer completes and that the receiver's state is changed to Running. It is likewise guaranteed by memory-integrity that a receiver's in-memory message buffer is changed only if the message transfer completes, which is the reason for the complexity of the arguments of the memory-integrity predicate above.

The additional object-integrity cases include a broadly symmetric case for receiving a message and resuming the sender, for resetting an endpoint and evicting a waiting sender or receiver, for updating the list of threads waiting at an endpoint, and for removing a virtual address space the subject owns from the active virtual address space database.

The cdt-integrity predicate limits all cdt changes to the subject's label.

The crucial property about integrity is transitivity:

Lemma 1. $\mathsf{integrity}\ p\ s_0\ s_1 \wedge \mathsf{integrity}\ p\ s_1\ s_2 \longrightarrow \mathsf{integrity}\ p\ s_0\ s_2$

This must be true at the top level for our statement about a single system call to compose over an execution which is a sequence of such calls.

Integrity is also trivially reflexive:

Lemma 2. $\mathsf{integrity}\ p\ s\ s$

3.5 Top Level Statements

Both integrity and pas-refined should be invariants of the system, which can be demonstrated using Hoare triples in a framework for reasoning about state monads in Isabelle/HOL. We have previously reported on this framework in depth [5]. In summary, the precondition of a Hoare triple in this framework is a predicate on the pre-state, the post condition is a predicate on the return value of the function and the post-state. In the case below, the return value is *unit* and can be ignored in the post condition. The framework provides a definition, logic, and automation for assembling such triples.

Theorem 1 (Integrity). *The property* integrity *pas st holds after all kernel calls, assuming that the protection state refines the policy, assuming the general system invariants* invs *and* ct-active *(current thread is active) for non-interrupt events, assuming the policy subject is the current thread, and assuming that the kernel state st is the state at the beginning of the kernel call. In Isabelle:*

$$\begin{array}{l}\{\!|\mathsf{pas\text{-}refined}\ pas \cap \mathsf{invs} \cap (\lambda s.\ ev \neq \mathsf{Interrupt} \longrightarrow \mathsf{ct\text{-}active}\ s) \cap \\ \quad \mathsf{is\text{-}subject}\ pas \circ \mathsf{cur\text{-}thread} \cap (\lambda s.\ s = st)|\!\} \\ \mathsf{call\text{-}kernel}\ ev \\ \{\!|\lambda\text{-}.\ \mathsf{integrity}\ pas\ st|\!\}\end{array}$$

We have shown in previous work [12] that the preconditions invs and $ev \neq$ Interrupt $\longrightarrow$ ct-active s hold for any system execution at kernel entry.

Theorem 2 (Authority Confinement). *The property* pas-refined *pas is invariant over kernel calls, assuming again the general system invariants* invs *and* ct-active *for non-interrupt events, and assuming that the current subject of the policy is the current thread.*

$$\begin{array}{l}\{\!|\mathsf{pas\text{-}refined}\ pas \cap \mathsf{invs} \cap (\lambda s.\ ev \neq \mathsf{Interrupt} \longrightarrow \mathsf{ct\text{-}active}\ s) \cap \\ \quad \mathsf{is\text{-}subject}\ pas \circ \mathsf{cur\text{-}thread}|\!\} \\ \mathsf{call\text{-}kernel}\ ev \\ \{\!|\lambda\text{-}.\ \mathsf{pas\text{-}refined}\ pas|\!\}\end{array}$$

We discuss the proof of these two theorems in the next section.

Via the refinement proof shown in previous work [12], both of these properties transfer to the C code level of the kernel. The guarantees that integrity *pas st* makes about user memory transfer directly, but the guarantees pas-refined *pas* and integrity *pas st* make about kernel-private state are mapped to their image across the refinement relation, which means we may lose some precision. The protection state of the kernel maps across the refinement relation precisely, the only difference between the model-level and C-level capability types being encoding.

The remainder of the system state does not translate so simply, but we contend that this does not matter. We envision the integrity theorem being useful mainly as a framing rule, with a component programmer appealing to the integrity theorem to exclude interference from other components and to the kernel model to reason about the component's own actions. In this case the programmer is interested not in the precise C state of the private kernel data, but about the related kernel model state. The integrity theorem will then provide exactly what is needed.

For proving confidentiality in the future, we may have to be more careful, because abstraction may hide sources of information that exist in the C system.

4 Proof and Application

4.1 Proof

The bulk of the proof effort was showing two Hoare triples for each kernel function: one to prove pas-refined as a postcondition, and one to prove integrity. These

lemmas are convenient to use within the Hoare framework as we can phrase them in a predicate preservation style. In the case of the integrity predicate, we use a form of the Hoare triple (the left hand side of the following equality) which encapsulates transitivity and is easy to compose sequentially. This form is equivalent to the more explicit form (the right hand side) as a consequence of reflexivity and transitivity:

$$\forall P.\ (\forall st.\ \{\!|\lambda s.\ \mathsf{integrity}\ pas\ st\ s \wedge P\ s|\!\}\ f\ \{\!|\lambda rv.\ \mathsf{integrity}\ pas\ st|\!\}) = (\forall st.\ \{\!|\lambda s.\ s = st \wedge P\ s|\!\}\ f\ \{\!|\lambda rv.\ \mathsf{integrity}\ pas\ st|\!\})$$

The proof was accomplished by working through the kernel's call graph from bottom to top, establishing appropriate preconditions for confinement and integrity for each function. Some appeal was made to previously proven invariants.

The proof effort generally proceeded smoothly because of the strength of the abstraction we are making. We allow the subject to change arbitrarily anything with its label, we map most kinds of access to the Control authority, and we require anything to which the subject has Control authority to share the subject's label. These broad brushstroke justifications are easy to apply, and were valid for many code paths of the kernel.

For example, CSpace updates always occur within the subject. As preconditions for various functions such as cap-move and cap-insert we assert that the address of the CSpace node being updated has the subject's label and, for pas-refined preservation, that all authority contained in the new capabilities being added is possessed by the subject in the policy. These preconditions are properties about the static policy, not the dynamic current state, which makes them easy to propagate through the proof.

The concept of a static policy gave us further advantages. By comparison, we had previously attempted two different variations on a proof that seL4 directly refines the take-grant security model [15]. These proof attempts were mired in difficulties, because seL4 execution steps are larger than take-grant steps. In between the take-grant steps of a single seL4 kernel call, their preconditions may be violated because capabilities may have moved or disappeared, and so the steps could not be composed easily. This seemed particularly unfortunate considering that the take-grant authority model has a known static supremum. Comparing against this static graph instead yields something like our current approach.

Another advantage we have in this proof effort is the existing abstraction from the C code up to our kernel model. The cdt (capability derivation tree) and endpoint queues described already must be implemented in C through pointer linked datastructures. In C the cdt is encoded in prefix order as a linked list. When a subject manipulates its own capabilities it may cause updates to pointers in list-adjacent capabilities it does not control. In the abstract kernel model the cdt is represented as a partial map from child to parent nodes, making all of our subject's CSpace operations local. It would be possible to phrase an integrity predicate on the C level which allowed appropriate pointer updates, but we think it would be extremely difficult to work with.

The combined proof scripts for the two access control properties, including the definitions of all formalisms and the SAC example, comprise 10500 lines

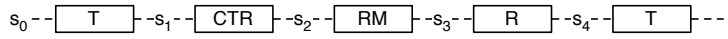
of Isabelle/HOL source. The proof was completed over a 4 month period and consumed about 10 person months of effort. Relative to other proofs about the kernel model this was rapid progress. Modifications to this proof have also been fast, with the addition of the cdt-integrity aspect of the integrity property being finished in a single day.

During the proof we did not find any behaviour in seL4 that would break the integrity property nor did we need to change the specification or the code of seL4. We did encounter and clarify known unwanted API complexity which is expressed in our policy wellformedness assumption. One such known problem is that the API optimisation for invoking the equivalent of a remote procedure call in a server thread confers so much authority to the client that they have to reside in the same policy label. This means the optimisation cannot be used between trust boundaries. An alternative design was already scheduled, but is not implemented yet.

We have found a small number of access control violations in seL4's specification, but we found them during a previous proof before this work began. These problems related to capability rights that made little sense, such as read-only Thread capabilities, and had not been well examined. The problem was solved by purging these rights.

4.2 Application

The application scenario of the integrity theorem is a system comprising trusted as well as untrusted components such as in the SAC example. Such scenarios are problematic for purely mandatory access control systems such as subsystems in a take-grant setting [15,7], because the trusted component typically needs to be given too much authority for access control alone to enforce the system's security goals (hence the need for trust). Our formulation of integrity enforcement per subject provides more flexibility. Consider a sample trace of kernel executions on behalf of various components in the SAC.



The example policy in Fig. 3 satisfies our wellformedness condition for the components R, T, and CTR. It does not satisfy wellformedness for RM. This means, given the initial state s_0, we can predict bounds for authority and state mutation using our theorems up to s_2 before RM runs. Since RM is the trusted component, we do not apply the integrity theorem, but reason about its (known) behaviour instead and get a precise characterisation of s_3. From here on, the integrity theorem applies again, and so on. Suitably constructed, the bounds will be low enough to enforce a system wide state invariant over all possible traces which in turn, if chosen appropriately, will imply the security goal.

As hinted at in Sect. 2, at state s_2, the RM component could use its excessive authority to reconfigure the system such that a new R is now connected to networks A and D. This setup would be incompatible with the policy applied to the transitions before, but a new policy reflecting the reconfiguration can be

constructed that applies from there on. If that new policy and the reconfiguration are compatible with the system invariant we can conclude the security goal for a system that dynamically changes its high-level policy even if the integrity theorem itself assumes a static policy per kernel event.

For reasoning about such systems, it is convenient to lift wellformedness and therefore pas-refined to sets of allowed subjects instead of a single current actor. This means the same instantiation of the theorem can be applied and chained over traces without further proof. We have formulated and proved the lifted version, but omit the details here. They add no further insight.

The set versions of integrity and pas-refined are also useful in a more restricted, but common scenario where the kernel is employed as a pure separation kernel or hypervisor. In this scenario, all components would be considered untrusted, and there can be one system-wide policy that is wellformed for the set of all components. Wellformedness can be shown once per system and the subjective integrity theorem collapses to a traditional access control formulation of integrity that applies to all subjects.

4.3 Limitations

The limitations of the theorem as presented mostly reflect API complexities that we circumvented by making assumptions on the policy.

The strongest such assumption is that we require two components with a Grant connection to map to the same label. This is no more than what a traditional take-grant analysis amounts to [7], but in our subjective setting there would be no strong reason to forbid Grant from trusted to untrusted components if the authority transmitted is within policy bounds. The difficulties with this and with interrupt delivery were discussed in Sect. 3.2. Trusted components can still delegate capabilities via shared CNodes, which appear in our graph as Control edges. This is the approach taken by the RM component of the SAC.

Another limitation is that the theorem provides relatively coarse bounds. This is required to achieve the level of abstraction we are seeking, but it is imaginable that the bounds could be too high for a particular frame condition that is required. In this case, one could always fall back to reasoning about the precise kernel behaviour for the event under consideration, but it was of course the purpose of the theorem to be able to avoid this.

5 Related Work

Security properties have been the goal for OS verification from the beginning: the first projects in this space UCLA Secure Unix [19] and PSOS [8] (Provably Secure OS) were already aiming at such proofs. For a general overview on OS verification, we refer to Klein [11] and concentrate below on security in particular.

A range of security properties have been proven of OS kernels and their access control systems in the past, such as Guttman et al's [9] work on information flow, or Krohn et al [13] on non-interference. These proofs work on higher-level

kernel abstractions. Closest to our level of fidelity comes Richards [17] in his description of the security analysis of the INTEGRITY-178B kernel in ACL2. Even this model is still connected to source code manually.

As mentioned, seL4 implements a variant of the take-grant capability system [15], a key property of which is the transitive, reflexive, and symmetric closure over all Grant connections. This closure provides an authority bound and is invariant over system execution. Similar properties hold for a broader class, such as general object-capability systems [16].

We have previously proved in Isabelle that the take-grant bound holds for an abstraction of the seL4 API [7,4]. The EROS kernel supports a similar model with similar proof [18]. However, these abstractions were simpler than the one presented here and not formally connected to code.

Compared to pure take-grant, our subjective formulation of integrity is less pessimistic: it allows certain trusted components, which are separately verified, to possess sufficient Control rights to propagate authority beyond that allowed by the policy.

6 Conclusions

In this paper, we have presented the first formal proof of integrity enforcement and authority confinement for a general-purpose OS kernel implementation. These properties together provide a powerful framing condition for the verification of whole systems constructed on top of seL4, which we have argued with reference to a case study on a real example system.

We have shown that the real-life complexity in reasoning about a general-purpose kernel implementation can be managed using abstraction. In particular, our formalisation avoids direct reasoning about the protection state, which can change over time, by representing it via a separate policy abstraction that is constant across system calls. Integrity asserts that state mutations must be permitted by this policy, while authority confinement asserts that the protection state cannot evolve to contradict the policy.

This work clearly demonstrates that proving high-level security properties of real kernel implementations, and the systems they host, is now a reality. We should demand nothing less for security-critical applications in the future.

Acknowledgements
We thank Magnus Myreen for commenting on a draft of this paper.

This material is in part based on research sponsored by the Air Force Research Laboratory, under agreement number FA2386-10-1-4105. 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. Andronick, T. Bourke, P. Derrin, K. Elphinstone, D. Greenaway, G. Klein, R. Kolanski, T. Sewell, and S. Winwood. Abstract formal specification of the seL4/ARMv6 API. http://ertos.nicta.com.au/software/seL4/, Jan 2011.
2. J. Andronick, D. Greenaway, and K. Elphinstone. Towards proving security in the presence of large untrusted components. In G. Klein, R. Huuck, and B. Schlich, editors, *5th SSV*, Vancouver, Canada, Oct 2010. USENIX.
3. D. Bell and L. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report MTR-2997, MITRE Corp., Mar 1976.
4. 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. Elsevier, 2009.
5. D. Cock, G. Klein, and T. Sewell. Secure microkernels, state monads and scalable refinement. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *21st TPHOLs*, volume 5170 of *LNCS*, pages 167–182, Montreal, Canada, Aug 2008. Springer-Verlag.
6. J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *CACM*, 9:143–155, 1966.
7. 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-Verlag.
8. R. J. Feiertag and P. G. Neumann. The foundations of a provably secure operating system (PSOS). In *AFIPS Conf. Proc., 1979 National Comp. Conf.*, pages 329–334, New York, NY, USA, Jun 1979.
9. J. Guttman, A. Herzog, J. Ramsdell, and C. Skorupka. Verifying information flow goals in security-enhanced linux. *J. Comp. Security*, 13:115–134, Jan 2005.
10. T. Jaeger. *Operating System Security*. Morgan & Claypool Publishers, 2008.
11. G. Klein. Operating system verification—an overview. *Sādhanā*, 34(1):27–69, 2009.
12. 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.
13. M. Krohn and E. Tromer. Noninterference for a practical DIFC-based operating system. In *IEEE Symp. Security & Privacy*, pages 61–76, 2009.
14. B. W. Lampson. Protection. In *5th Princeton Symposium on Information Sciences and Systems*, pages 437–443, Princeton University, Mar 1971. Reprinted in Operat. Syst. Rev., **8**(1), Jan 1974, pp 18–24.
15. R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *J. ACM*, 24(3):455–464, 1977.
16. T. Murray and G. Lowe. Analysing the information flow properties of object-capability patterns. In *6th FAST*, volume 5983 of *LNCS*, pages 81–95, 2010.
17. R. J. Richards. Modeling and security analysis of a commercial real-time operating system kernel. In D. S. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 301–322. Springer-Verlag, 2010.
18. J. S. Shapiro and S. Weber. Verifying the EROS confinement mechanism. In *IEEE Symp. Security & Privacy*, pages 166–181, Washington, DC, USA, May 2000.
19. B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verification of the UCLA Unix security kernel. *CACM*, 23(2):118–131, 1980.

University of New South Wales
School of Computer Science and Engineering

Formal Functional Specification of a Security Critical Component in Isabelle/HOL

Author: Nelson Billing

Supervisor: June Andronick

Assessor: Gerwin Klein

October 18, 2011

A Thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Engineering (Software Engineering)

Acknowledgements

I would like to thank the following persons: Ning Yu, June Andronick, Gerwin Klein, Thomas Sewell, David Greenaway, Bernard Blackham, Rafal Kolanski, David Cock, and Matthias Daum. This list is not intended to be exhaustive, and is in no particular order.

Contents

1 Introduction **6**

2 Background **9**

2.1 Formal verification . . . 9
2.2 The seL4 Microkernel . . . 10
2.3 seL4's Formal Proof of Correctness . . . 11
2.4 The Secure Access Controller . . . 12
2.5 The Router Manager . . . 14
2.6 Access control in seL4 . . . 15

3 Design **16**

3.1 Router manager design . . . 17
3.2 libseL4 design . . . 18
3.2.1 ARM & x86 . . . 19
3.2.2 Generic libseL4 . . . 19
3.3 System process design . . . 19
3.3.1 System process from seL4 refinement . . . 19
3.3.2 Modifications . . . 20
3.3.3 Generic system process . . . 21

4 Implementation **22**

4.1 Router Manager Functional specification . . . 22
4.1.1 The *sac_abstract_state* . . . 23
4.1.2 Router Manager program instructions . . . 24
4.1.3 Router Manager program structure . . . 26
4.1.4 Example: *Teardown Router* . . . 27
4.1.5 Example: *Setup Router* . . . 28

4.1.6 In-depth *instr* examples . . . 29
4.1.7 Internal helper function examples . . . 32
4.2 libseL4 specification . . . 33
4.2.1 The *tc_monad* . . . 34
4.2.2 The *thread_abstract_context* . . . 34
4.2.3 libseL4 specification structure . . . 34
4.2.4 In-depth libseL4 examples . . . 35
4.2.5 libseL4 helper function examples . . . 37
4.2.6 Lifting from *tc_monad* to *rm_monad* . . . 38
4.3 System process . . . 39
4.3.1 The *global_automaton* . . . 39
4.3.2 The *ADT_I* . . . 41
4.3.3 Kernel specification . . . 42
4.3.4 User specification . . . 42

5 Conclusions **44**
5.1 Contributions . . . 44
5.2 Future Work . . . 44

A Router Manager Functional Specification **46**
A.1 The Router Manager Functional specification program . . . 46
A.2 The sac_abstract_state . . . 47
A.3 The rm_monad and Router Manager execution . . . 48
A.4 Router Manager helper functions . . . 49
A.5 Router Manager instructions . . . 50
A.6 Router Manager program . . . 57

B libseL4 Specification **61**
B.1 libseL4 Interface . . . 61
B.2 seL4 Interface . . . 65
B.3 libseL4 Functions . . . 66
B.4 libseL4 Types . . . 66

C System Process **73**
C.1 Router Manager Functional specification system process . . . 73
C.1.1 Lifting from rm_monad to user_state transitions . . . 73
C.2 User state translations . . . 74
C.3 The user specification . . . 75

C.4 The system process 75
C.5 System process 75
C.6 The global_automaton 76

Chapter 1

Introduction

Computers are extremely pervasive today, and so of course is software. Software is notoriously difficult to get right, for the most part developers just accept uncertainty in a piece of software to some degree.

In software applications with very little fault tolerance, the assurance given by standard software testing is unsatisfactory and so there are a number of approaches to achieving higher levels of assurance. One obvious approach is just to do more testing, the more testing you do the closer to complete coverage of inputs and therefore less likely that bugs will slip through.

Alternatively there are formal verification and static code analysis techniques which can deliver stronger claims about correctness but which are typically more costly than testing. Therefore formal methods techniques are applied where the need for assurance justifies the cost.

The goal of the trustworthy embedded systems project at NICTA, where this thesis project is taking place, is to build systems we can really trust, by providing formal proofs about their security, safety and reliability. The first step in this vision was the creation of the seL4 microkernel, a general purpose microkernel of the L4 family, along with a formal proof of its functional correctness[3]. The proof of functional correctness was completed in 2009 and since then work has been going on to build larger systems on top of the seL4 microkernel, and prove specific, targeted properties about those systems. This thesis project is a part of these next steps of the vision. The idea being how can we use this verified kernel to prove properties about user programs?

The Secure Access Controller (SAC) is a case study in verification of systems built on seL4. The role of the Secure Access Controller is a secure router between two networks which must be kept isolated- meaning that the targeted property we want to hold about the SAC is absence of data flow between the networks.

The trustworthy embedded systems project already have a very high level security model of the

SAC and a proof of absence of data flow on that model[1]. The aim of the larger project, which this thesis is a part of, is connecting the SAC Security model to the implementation of the SAC so that the proofs on the Security model hold for the source code as well. Connecting the two up means showing that the SAC source code has the same behaviour as the security model used for the proof.

We decided to show refinement via an intermediate step. The SAC's architecture consists of only one trusted component, called the Router Manager and, as we will show, the behaviour of *untrusted* components can safely be underspecified as arbitrary for the purposes of verification. Therefore the intermediate step would be a Functional specification of the Router Manager component of the SAC.

The Router Manager component must make calls into the seL4 kernel to perform its function, such as to create and setup new threads, and to receive control messages. In the C implementation of the Router Manager component, calls to the kernel are made by calling functions from a C library built for seL4, called libseL4. Instead of effectively 'inlining' the libseL4 code into the Router Manager Functional specification, we would like to implement a libseL4 specification. It is desirable for us to implement the libseL4 specification such that it can be used in other seL4 user program specifications. For this reason, we stress that the libseL4 specification must be *generic*.

With the Functional specification done, we would need a framework for proving the new Functional specification refines the Security model . The framework would preferably be general in the specification of user behaviour so that it can be reused in showing refinement between other sel4 user program specifications. In other words, we would like to be able to plug any user program specification into the system process framework and reason about the resulting system process.

Writing the Router Manager Functional specification was the main objective of this thesis, with the aim of including a generic libseL4 specification. A stretch goal was to start on a framework for the refinement proof.

We completed the Functional specification, which involved specifying a subset of the libseL4 library alongside it. The libseL4 subset was implemented in a generic fashion so that it can be used in specifying other user programs (and we would expect that most or all other seL4 user programs do use libseL4 in their C implementation). We also implemented the framework for stating refinement between seL4 user programs, along with an example of its use with the Functional specification.

This document firstly presents the necessary background in formal verification, seL4, and the Secure Access Controller in Chapter 2. Following that is Chapter 3 which presents the design of the Router Manager Functional specification, libseL4 specification, and refinement framework, with

the aim of explaining the choices in implementation and providing some much more detailed background.

Chapter 4 is structured around the three main parts of the project that were implemented. In each section, details from the Isabelle/HOL theories are presented and discussed in order to show the reader the 'hows' of the implementation. The final chapter of this document, Chapter 5, summarises the contributions made in this project and offers the experience gained during the work, as well as what remains to be done in future work.

Chapter 2

Background

This chapter is intended to cover some general background on topics related to this thesis project which may be helpful to readers. Section 2.1 gives an overview of formal methods techniques, the particular technique used in this thesis, and a justification of our choice. Next, Section 2.2 will introduce the seL4 microkernel and give some background for readers who are unfamiliar. Section 2.3 introduces the techniques used in the seL4 formal verification project which guided the approach in this project. After that, Section 2.4 presents some background on the target of the specification project, the Secure Access Controller, its components and its design. As a followup, Section 2.5 briefly discusses the central component of the SAC, the Router Manager. The last section of this chapter, Section 2.6, gives some background information on how access control is handled in seL4 and how this guides the componentization of the SAC.

2.1 Formal verification

There are several broad approaches to formal verification with different levels of expressiveness- in terms of the properties about programs- and automation.

Model checking, where we create a model of our system and then use an automated tool to explore its state space, gives us a lot of expressiveness in talking about properties of a program. While verification is fully automatic, it is not feasible for all programs because of the so-called *state explosion* problem. Besides, the technique does not involve showing refinement from source code to the model, so it does not prove anything directly about the source code.

Static code analysis techniques can be used to automatically detect patterns of programming mistakes- like memory leaks or out-of-bounds memory access- so that specific strong claims can be made about programs. The nature of automatically checking source code for 'mistakes' does

not lend itself to proving arbitrary program-specific properties.

Theorem proving enables us to reason formally about a logical model of the program that can be strongly linked to the program itself using a formal semantics of the programming language of the program. This technique allows us to phrase arbitrary properties about a program and prove them at the source code level. Unlike model checking, state space is not immediately important because we are not bound to prove by exhaustion. The price we pay with theorem proving is that each stage of the process requires large amounts of human work along with mechanical assistance.

The formal verification technique which was chosen for this thesis project is theorem proving. The reasons for this choice are the same reasons that it was chosen for the seL4 verification. First the large state space of seL4 and the SAC makes automatic state space exploration impractical. Second, our goal is a code level guarantee. And finally, we are aiming at specific properties that need to be phrased in an expressive language.

Verification of the seL4 kernel was carried out using the Isabelle theorem prover [6], and the models of the seL4 kernel are written as Isabelle/HOL theories. Since this thesis project aims at leveraging the proofs of the seL4 kernel, it is natural to continue using Isabelle/HOL, which also happens to be the theorem prover I have experience with.

2.2 The seL4 Microkernel

The seL4 microkernel is a L4 family microkernel, with capability-based security [2]. seL4 is the first general purpose OS kernel with formal proof of its functional correctness.

seL4 provides the usual L4 abstractions: threads, address spaces, and inter-process communication (IPC). Threads, address spaces, IPC and all other (more esoteric or architecture specific) abstractions are represented by *kernel objects*. Authority over *kernel object*s is conferred via capabilities. In fact, all kernel calls in seL4 are invocations on a capability- for instance mapping a frame into an address space is an invocation on a capability to the frame.

Capabilities in seL4 serve a similar function to Linux file handles. However they are meant to implement much stricter access control policy. Capabilities are kept *for* threads, in their respective *CSpaces* which are conceptually similar to address spaces. The capabilities themselves are unforgeable and threads are only ever given pointers to the *CSpace* and then into *slots* inside it, so as to prevent any tampering with the capability kernel object. There are strict controls on granting capabilities over IPC, and threads may only copy or move capabilities between or inside *CSpace*s indirectly via the kernel.

A *CSpace* in the kernel is made up of a hierarchy of *CNodes*. A *CNode* is a kernel object which

serves as a container for capabilities. The hierarchy is built by placing capabilities to one *CNode* inside another *CNode*, thereby allowing addresses to be 'walked' from a root *CNode*, through subsequent *CNodes* to some particular capability in that *CSpace*.

Rights to create new kernel objects in system memory are represented by *Untyped* capabilities, each having a start address and size to indicate which region of memory it covers. When a user wants to create a new kernel object, they must *retype* some part of an *Untyped* capability they hold to do so.

As kernel objects or new capabilities are created using the authority of a capability, the kernel keeps track of which objects and capabilities are *children* of that capability. The counterpart to *retyping* an *Untyped* capability is to call *revoke* on that capability. When *revoke* is performed on the original *Untyped* capability, all of the *child* capabilities are also revoked, and then destroyed. Any kernel objects associated with these capabilities are destroyed in the process.

Of course, when we speak of the user performing some action at the kernel objects or capabilities, or performing inter-process communication, what we mean is that the user is asking the kernel to perform this action on their behalf. Readers will be familiar with Unix syscalls and filehandles, and seL4 syscalls and capabilities roughly correspond to these respectively.

2.3 seL4's Formal Proof of Correctness

The seL4 microkernel makes a part of the trusted computing base for the SAC, and in that sense this project is building directly on the formal verification of functional correctness of seL4. Functional correctness here meaning that the source code of seL4 implements the desired specification. The process of verifying seL4 is well documented and [5][3][7] are instructive to the techniques used.

The first step in proving functional correctness is to write the formal specification of the program. This describes *what* the kernel is supposed to do, whereas the implementation describes *how* it does it. In this case the formal specification of the seL4 kernel is an Isabelle/HOL theory termed the *abstract specification*. The abstract specification has an abstract view of the seL4 kernel state and is nondeterministic. It is written in the functional style of Isabelle/HOL programs and uses nondeterministic and deterministic state monads.

The next step is to prove that the source code of the program actually implements the specification. The way this was done for seL4 is the *implementation*, which is a translation of the seL4 source code into Isabelle/HOL, is shown to be a refinement of the *abstract specification*.

Since the refinement relation is transitive the refinement from the *implementation* to the *abstract specification* can be completed in two steps: from the *implementation* to an *executable specification*

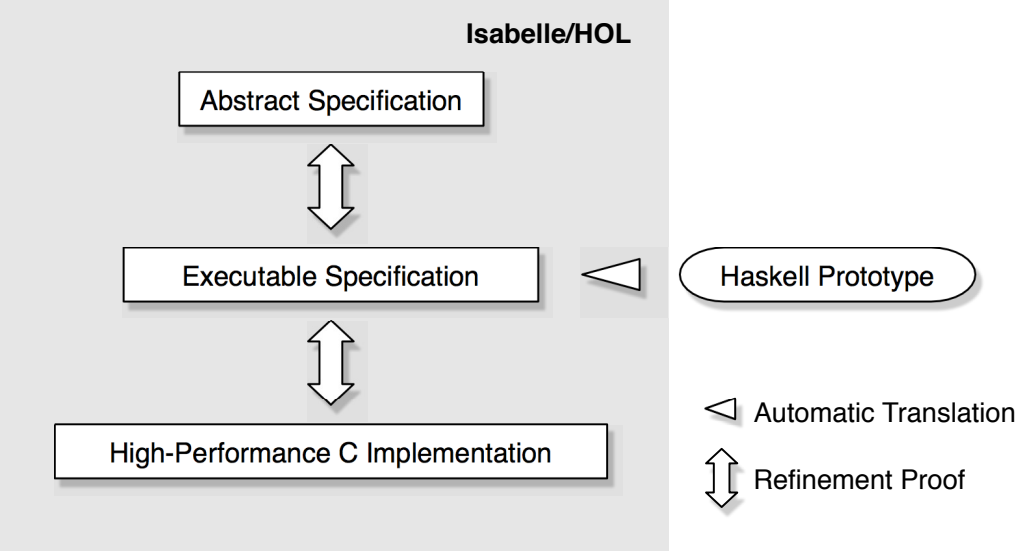


Figure 2.1: Refinement steps in seL4 refinement [3]

and from the *executable specification* to the *abstract specification*, as illustrated in Figure 2.1.

A concise argument for the soundness of the techniques used in the refinement is given in [5]. Each of the two refinement steps is proved by a forward simulation between two of the processes, the general shape of these forward simulations is illustrated in Figure 2.2. The idea of forward simulation is to show that given a *state relation* between two processes, when the processes are started in some related state and react to the same event they will end in some related states. Forward simulation is proven by showing an equivalent property termed *correspondence*, which is defined between terms in the specifications rather than *event*s.

2.4 The Secure Access Controller

Ultimately this project is a case study in proving correctness of a software system. The system used is a Secure Access Controller (SAC), a router between a trusted user terminal and a number of untrusted networks, as illustrated in Figure 2.3. For simplicity and without loss of generality we will only consider two such untrusted networks.

The best way to explain the role of the SAC is to imagine a user who needs access to two different computer networks. The user is not allowed to connect to both networks at once. The administrators of these two networks trust the user not to save information to pass it on, but do not trust his network card/TCPIP stack/OS to keep their network isolated from other networks. The role

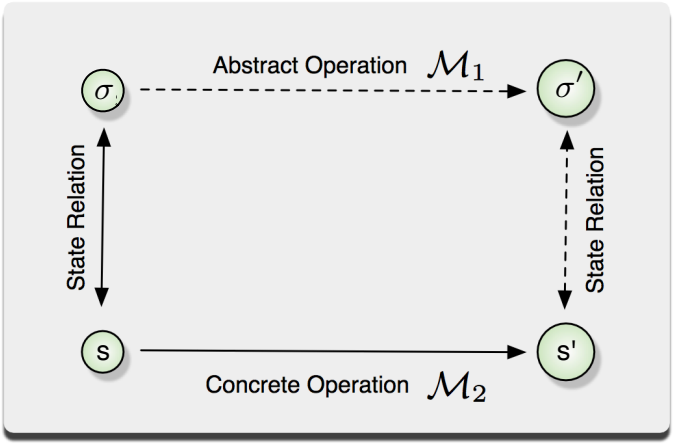


Figure 2.2: Forward simulation [3]

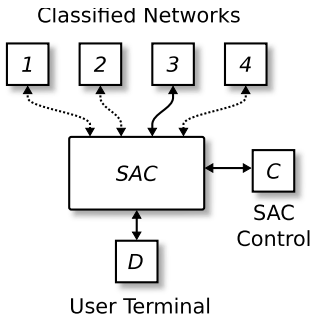


Figure 2.3: The SAC routes between a user's terminal and 1 of n classified networks [1]

of the SAC here is to sit between the user terminal and the two outside networks and connect the user to one network at a time. To satisfy the administrators we need to show that the SAC won't leak data between the networks.

The SAC needs a component responsible for network routing, including network card drivers. This component will be called the Router. It also needs a web interface where the user can request a network switch, this will be called the SAC Controller. These two components represent a significant amount of code, much larger than seL4. Proving non-trivial properties about that much source code is infeasible, so the guiding design principle of the SAC was a minimal trusted computing base (TCB) in order to simplify the proof process.

Our system of course needs to trust the behaviour of the seL4 microkernel- for which we have a proof- but the SAC program can also be split into several *components*, some being in the trusted

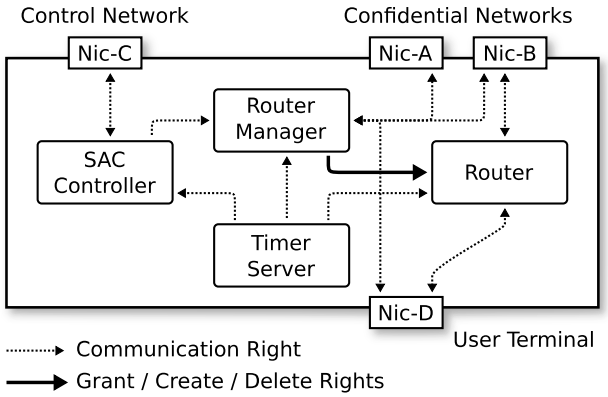


Figure 2.4: High level component breakdown of the SAC design [1]

computing base and others not. The logical way to split the program into components is to consider the least privilege each part of the program needs in order to function, so that components individually have a small subset of the capabilities that the SAC has as a whole.

The targeted security property we focus on in this thesis project is absence of data flow between networks, so the idea is to pinpoint components which can access both networks. For example, the Router is given access to network interfaces to the user terminal and *only one* of the outside networks. It was implemented by a Linux instance comprising millions of LOC. Therefore we place it outside of our TCB. To ensure we do not need to trust the Router we destroy it whenever we switch from one outside network connection to another. This critical operation is done by a trusted component, called the Router Manager.

The Router Manager has complete access to both external networks and thus is able to break our security property. This places the Router Manager inside our TCB. The remaining components are the SAC Controller, which is given access only to the control network, and a timer server. As we can see in Figure 2.4, only the Router and Router Manager components have access to the outside networks. Therefore given some reasonable assumptions about seL4 enforcing access control, discussed in Section 2.6, the only trusted component we need to verify is the Router Manager.

2.5 The Router Manager

The Router Manager (RM) component of the SAC system is the core logic of the system and the one *trusted* component. The Router Manager sits in an infinite loop and waits for instructions from a Control Network interface. Once it receieves a message it can either just teardown the

Router component of the SAC or teardown the Router component and then setup a new Router to either Network A or Network B. Once the Router Manager has finished responding to the control message it will return to waiting for another message. This process is discussed in more detail in Subsection 4.1.3.

Setting up the Router component involves creating a Router *CSpace*, a Router virtual address space, a Router Inter-Process Communication buffer, a Router Thread Control Block, and then connecting them all together and activating the new thread. Regions of memory such as the Router's program text and some free space need to be mapped into the Router's virtual address space. Additionally, the capabilities to either the Network A interface or Network B interface, and some *Untyped* capabilities, need to be transfered into the Router's *CSpace*. This process is explained in more depth in Subsection 4.1.5.

Tearing down the Router component involves calling *revoke* (see Section 2.2) on all *Untyped* capabilities which were used in setting up the Router, as well as 'sanitising' whichever network interfaces the Router was connected to (Networld D and either Network A or Network B). This process is explained in more depth in Subsection 4.1.4.

2.6 Access control in seL4

In order to know that the untrusted SAC components cannot interfere with our security property we need to know that seL4 is enforcing an access control policy on these processes. For example we need to know that the seL4 kernel will not allow the Router to access both outside networks with just a capability to one, or move capability pointers around in the Router Manager's CSpace.

Showing that seL4 enforces this access control policy is ongoing in the verification team at NICTA. One half is shown in the recently completed Integrity theorem [4], while the remaining half is the work in progress Confidentiality theorem. Roughly, the Integrity theorem shows that processes are bounded in writes and Confidentiality is intended to show that processes are bounded in reads.

Chapter 3

Design

The aim of this chapter is to step through the design of the Router Manager Functional specification, as well as the system process that we want to be a combination of the kernel and Router Manager. The Router Manager Functional specification is meant to specify the behaviour of the Router Manager component of the Secure Access Controller, that is, it gives the set of all seL4 user state transitions that the Router Manager can make.

The Router Manager needs to perform system calls in setting up or tearing down other processes or receiving messages. In the C implementation of the Router Manager the system calls are done via the libseL4 library and so it is necessary to specify some subset of libseL4 in order to have a full specification of the Router Manager that can interact with a specification of the seL4 kernel.

With libseL4 giving the interface to the kernel, we also need a kernel specification, so that the system process can 'trap' into the kernel after a syscall is set up by the user process. Specifications of the seL4 kernel at varying levels of abstraction already exist.

The final piece of the puzzle in defining the system process is a framework that will combine the transitions of the user specification (the Router Manager Functional specification) and the seL4 kernel. This requires definitions to translate or 'lift' between views of system state and a definition of the overall system transitions given the user and kernel transitions. The latter is refered to in this document as an *automaton*.

This chapter first discusses the design of the Router Manager in Section 3.1, then the design of libseL4 in Section 3.2, and finally Section 3.3 presents the design of the system process.

3.1 Router manager design

The purpose of the Router Manager Functional specification, is to provide a less jarring refinement step for both the security model and implementation. Thus the suitable level of abstraction for operations and data in the Router Manager Functional specification would lie somewhere between the implementation and security model. To contrast with the verification of seL4: the high level security model of the SAC is far more abstract than the *abstract specification* of the seL4 kernel, mentioned in Figure 2.1. seL4's *abstract specification* is a full functional specification of the kernel, whereas the SAC's security model abstracts away everything that is not needed for the security analysis. The Router Manager Functional specification is intended to employ a similar level of abstraction to the seL4 *abstract specification* and describe the RM's functional behaviour, as shown in Figure 3.1.

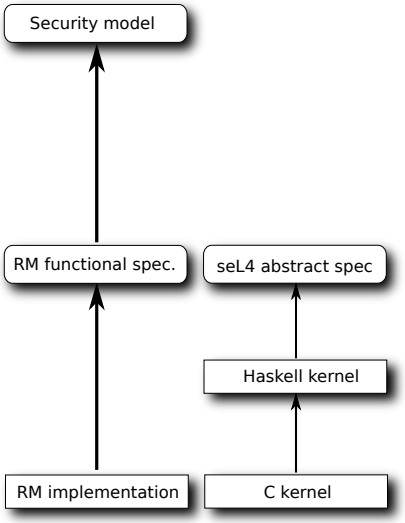


Figure 3.1: Side-by-side comparison of SAC and seL4 refinement artefacts.

The Router Manager Functional specification of course will need a program state capturing the state we want to project out of the security model and implementation's states. The content of the state contains most of the internal state of the RM implementation in a more abstract form.

There is a fundamental mismatch between the security model's RM program and the RM implemention in that the first is unstructured code, branching with jumps, while the second has the program structure of a C program- *if*, *for*, *while*, etc. A middle ground between structured and unstructured code is unhelpful, so the Router Manager Functional specification represents programs as unstructured code.

```
datatype sysOPs =
       SysRead        cap
     | SysWrite       cap bool
     | SysCreate      cap
     | SysGrant       cap cap
     | SysRemoveSet   cap (cap set)
     | SysDelete      cap
```

Figure 3.2: Definition for kernel calls in Router Manager security model.

RM steps can be divided into internal processing and calls into the seL4 kernel. In the security model kernel calls are represented by an abstract instruction datatype, shown in Figure 3.2, and the only internal processing is the jump instructions which move control to some other position in the program non-deterministically.

The RM implementation takes mostly internal steps punctuated by kernel calls to set up or communicate with other processes. Details on how the Router Manager Functional specification has been defined are given in Chapter 4. The kernel calls involve setting up registers and trapping into the kernel; this process is wrapped in libseL4 library functions which the RM's implementation uses, and which are discussed in the next section.

3.2 libseL4 design

As mentioned in Section 3.1, the router manager program uses the libseL4 library as an interface with the seL4 microkernel. Therefore, to fully specify the router manager program we need a specification of the libseL4 library. The router manager needs to cross the kernel boundary to create a new thread, transfer capabilities, retype untyped memory into new kernel objects, or to map memory into another address space et cetera, so all of these tasks will involve executing code from libseL4. Unfortunately no such specification existed, requiring me to specify the parts of libseL4 that are used by the router manager.

The libseL4 library provides a simple interface to the kernel, as it is described in the seL4 manual, in terms of a set of '*invocations*' for each capability type. Internally the kernel decodes calls made to it and then becomes aware of what the *invocation* it is handling, but the user side of the interface does not have functions for each *invocation*, and arguments to the kernel calls and magic numbers must be put into registers for the kernel to decode. Thus libseL4 is useful as providing an abstraction of the real calling mechanism, as well as marshalling and copying data that the user passes as arguments.

The implementation of the libseL4 specification is presented in detail in Section 4.2.

3.2.1 ARM & x86

The architecture specific code in the router manager comprises just calls to the kernel interface functions, so selecting the architecture that the router manager specification would assume is essentially choosing the version of libseL4 to use. Since the existing router manager program was written for the x86 architecture whereas an ARM- but not x86- specification of the seL4 kernel exists, I had to make a choice between the two architectures for my libseL4 specification.

Inevitably I chose to specify the ARM libseL4 implementation, since the seL4 kernel specification is overwhelmingly larger and more complex than the router manager program. Before the refinement from the Router Manager Functional specification to the C code, the router manager C code will need to be ported to ARM. Given the small amount of architecture specific code in the router manager this should not alter the overall structure of the program.

3.2.2 Generic libseL4

Most of the motivation for this project lies in completing a case study in verifying software, specifically what sort of leverage we can net by designing that software as a seL4 user program. With that in mind, developing the specification for a subset of the libseL4 library needed to be done in such a way that it would be agnostic to a user program specification importing it. This way any other seL4 user program could be specified in the fashion most appropriate to that program, and where it would interface with the kernel the libseL4 specification could describe that behaviour.

Initially I identified two key features that a *generic* libseL4 specification would need. Firstly the state that libseL4 functions operate over would need to be a subset of any seL4 user program's state. Secondly the libseL4 specification should introduce all the types that the specification of a seL4 user program might use when communicating with seL4.

3.3 System process design

Given the Router Manager security model and the functional specification we have most of the components we need to begin showing that the C code for the router manager refines the security model, but still remaining is the framework for showing either of the refinement steps.

3.3.1 System process from seL4 refinement

As discussed in Section 2.3 the method of program verification for the seL4 kernel was refinement from an abstract specification to source code, via an *executable* specification. In that case

refinement was shown by defining a process, representing the kernel and user reacting to events, for each kernel specification. What is shown, then, is that the behaviour of one process refines the other.

To this end, a somewhat generic kernel-system automaton already exists. Roughly speaking; a non-deterministic kernel specification can be given which dictates *kernel mode* behaviour, and *user mode* behaviour is unspecified. The three *modes* under which the system operates are shown in Figure 3.3. Separate kernel and user state is recognised, where the user state will be a projection on the kernel state.

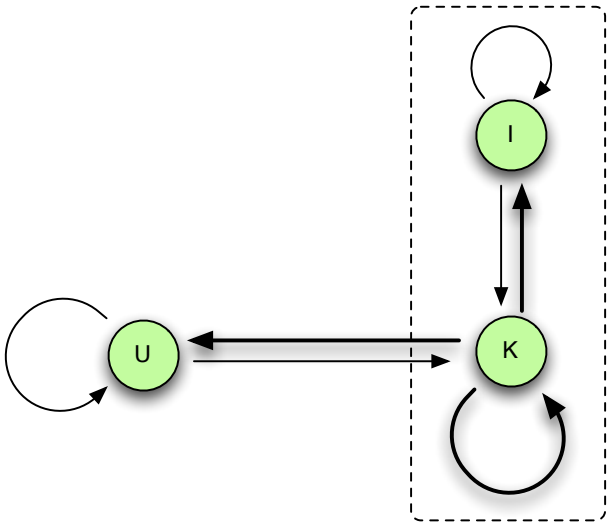


Figure 3.3: System transitions between *Idle*, *User*, and *Kernel* modes.

3.3.2 Modifications

In adapting this framework for the Router Manager refinement there were a number of immediate questions: which kernel specification would be used, should the user state be expanded, how should the kernel react to events, and should the automaton treat a kernel to user transition differently than a user to user transition?

The original intention in designing the RM Functional specification was to try for a level of abstraction similar to that in the seL4 Abstract specification, therefore it seemed logical to choose the seL4 Abstract specification as the kernel specification in the RM Functional specification system process.

User state needed to be expanded so that an abstract program counter called the *instruction pointer*

could be maintained for the active user process when the system is in *user mode*. The *instruction pointer* is necessary so that the user specification can control its own program flow and resume at the correct place when returning from a kernel call or being scheduled.

The system process updates the abstract *instruction pointer* on a *kernel mode* to *user mode* transition, using the program counter placed in a register by the kernel. A user specification is expected to update its *instruction pointer* in transitions, except when setting up a software interrupt since the kernel will update the program counter in a register when servicing the kernel call. This also answers the fourth system process design question above.

3.3.3 Generic system process

Being that this project is aiming to test-run seL4 user program verification, it is important that the newly modified system process is *generic* enough that it can represent any user system running on top of seL4. Specifically it should have a view of user state which will be a superset of any user program's observable state. The generic system process framework implementation is discussed in Section 4.3.

Chapter 4

Implementation

The purpose of this section is to describe in detail how it is that the Router Manager Functional specification was implemented. As discussed in Chapter 1, the original goal of this thesis project was functional specification of the Router Manager program. This goal is complete, and the details of its implementation are contained in Section 4.1.

Also discussed was that a stretch goal, or future work, from this project could be specifying the libseL4 library and/or creating a framework for proving refinement between the Router Manager Security model and the Functional specification. The parts of the libseL4 library which are used by the Router Manager program are specified, and in a generic manner which would allow them to be used, covered in Section 4.2. The foundation of a generic user refinement framework will be a user and kernel system automaton, I implemented this as well, covered in Section 4.3.

4.1 Router Manager Functional specification

The aim of the Router Manager Functional specification is to create a specification for the Router Manager which is more abstract than the Router Manager C implementation, but more concrete than the Secure Access Controller Security model. Structurally we attempted to mirror the C implementation and Security Model as often as possible.

This chapter will present the implementation of the Router Manager Functional specification. Subsection 4.1.1 discusses the Functional specification's state, Subsection 4.1.2 presents the way program transitions are represented in the Functional specification, Subsection 4.1.3 gives the structure of the Router Manager program, and the remaining subsections give examples from the Functional specification's Router Manager program.

The examples in the later subsections start from an instruction in the top level Functional spec-

ification Router Manager program and then expand more definitions in subsequent subsections until they reach the kernel barrier. Then the examples in Section 4.2 regress the same tasks further. The entire implementation of the Router Manager Functional specification is too big to discuss in detail in this report, but it is given in full in Appendix A.

4.1.1 The *sac_abstract_state*

```
record sac_abstract_state =

  ip :: seL4_Word

  untyped_items :: seL4_CPtr list list
  free_items :: seL4_Word list
  total_items :: seL4_Word list

  info :: seL4_MessageInfo
  message :: seL4_Word
  var_i :: nat

  RM_RWGC_NIC_A_cap :: seL4_CPtr
  RM_RWGC_NIC_B_cap :: seL4_CPtr
  RM_RWGC_NIC_D_cap :: seL4_CPtr

  client_vspace :: seL4_CPtr
  virtualirq_endpoint :: seL4_CPtr
  command_endpoint :: seL4_CPtr
```

Figure 4.1: Definition of the *sac_abstract_state*

The *sac_abstract_state* is an abstract representation of the Router Manager's user memory view. In terms of the Router Manager C program, this would correspond to allocated memory, local, and global variables. As mentioned in Section 3.3, the Router Manager program is expected to update its own *instruction pointer*, so this is contained inside the *sac_abstract_state*.

Figure 4.1 shows the *sac_abstract_state* global and local variables from the C implementation of the Router Manager. *ip* is the abstract *instruction pointer*, *info*, *message*, and *var_i* are local variables. The remaining ones are global variables.

untyped_items contains pointers to all of the *Untyped* capabilities the Router Manager has, *free_items* contains how many *Untyped* capabilities of each size are free, and *total_items* contains how many of each size *Untyped* the Router Manager has in total.

*RM_RWGC_NIC_*𝕀*_cap* contains a pointer to the Router Manager's all-rights (**R**ead **W**rite **G**rant **C**reate) capability to Network 𝕀 interface.

client_vspace, *virtualirq_endpoint*, and *command_endpoint* are pointers to capabilities which are set up by a booter initially and which we just pass around.

4.1.2 Router Manager program instructions

The Router Manager program is represented by a list of instructions. The instructions are themselves defined as functions across the Router Manager's abstract state and optionally producing an *event* indicating if a kernel event was generated by the instruction. The type of instructions is *(event option) rm_monad*.

The *rm_monad* type represents a computation which can modify the RM program state, shown in Figure 4.2. *nondet_monad* here is from some non-deterministic monad infrastructure written by Gerwin Klein and is defined in the usual way, with *bind*, *return* and sugar like *do*, *;* and $\leftarrow$. Functions which modify the RM program state can be defined independently of an *event* and then used in the definitions of different instructions. Functions which do not modify the RM program state- that would be so-called *pure* functions- can be defined as ordinary Isabelle/HOL functions over their input and output types.

As shown in Figure 4.2 the *rm_monad* state comprises a *sac_abstract_state* and *thread_abstract_state*. The former being the state which is considered specific to the Router Manager program implementation, and the latter being the state which is common to any seL4 user program. The reason that a clean separation between the two is desirable will be explained in Section 4.2. The *thread_abstract_context* will be explained in Subsection 4.2.2.

```
types   'a rm_monad  =   ((thread_abstract_context × sac_abstract_state), 'a) nondet_monad
types   instr        =   (event option) rm_monad
types   program      =   instr list
```

Figure 4.2: Definition of program terms in Router Manager functional specification.

rm_monad is a non-deterministic state monad. This is because, while the Router Manager Functional specification *is* deterministic, there is no reason for the generic seL4 user program verification machinery to assume determinism. In fact, the SAC Security model is non-deterministic, and so this choice also means we can use the same machinery to build a process for the SAC Security Model.

For computations of the type *unit rm_monad*, we wrap the non-event-generating computation in a function to increment the *instruction pointer* and return *None*, signifying no event.

As shown in Figure 4.3 the *k_instr* wrapper is the identity and is included to balance the *u_instr* wrapper, where *k_instr* is used on instructions which trap into the kernel and *u_instr* is used on instructions which do not. Instructions which trap into the kernel do not need to increment the

instruction pointer, as the kernel will do this, nor do they need to be wrapped in a *None*, since they are guaranteed to return *Some (SyscallEvent sy)*.

```
definition k_instr ::
  instr ⇒ instr
  where
  k_instr = id
```

```
definition u_instr ::
  'a rm_monad ⇒ instr
  where
  u_instr m =
     do m;
        incr_pc;
        return None
     od
```

Figure 4.3: *u_instr* and *k_instr* definitions.

The definition of an *event* in Figure 4.4 comes from the seL4 *Executable specification* (see Figure 2.1) of the seL4 microkernel. This type captures all of the possible events that the seL4 kernel responds to. For the purposes of the Router Manager functional specification we need only consider *SyscallEvent*, since a user program will cannot intentionally spawn the other events. On ARM- our chosen architecture- this corresponds to a software interrupt.

```
datatype event =
     SyscallEvent syscall
   | UnknownSyscall nat
   | UserLevelFault machine_word machine_word
   | Interrupt
   | VMFaultEvent vmfault_type
```

Figure 4.4: Definition of *event*.

Again, the definition of *syscall* in Figure 4.5 comes from the seL4 *Executable specification*. The Router Manager functional specification makes libseL4 calls which only use *SysCall* and *SysWait*, however a full libseL4 specification would potentially use all of them. For clarity: every Router Manager functional specification instruction returns either *None*, *Some (SyscallEvent SysCall)* or *Some (SyscallEvent SysWait)*.

```
datatype syscall =
     SysSend
   | SysNBSend
   | SysCall
   | SysWait
   | SysReply
   | SysReplyWait
   | SysYield
```

Figure 4.5: Definition of *syscall*.

As discussed in Section 3.1, the Router Manager program is an unstructured list of instructions. Branching and conditional execution in the Router Manager program is achieved by 'jumping', we define a *jmp* and *cnd_jmp* in Figure 4.6.

```
definition jmp ::
   seL4_Word ⇒ instr"
  where
  jmp n =
     do modify (λ(tc, rm). (tc, ip_update (λx. x + 1) rm));
        return None
     od"


definition cnd_jmp ::
  ((thread_abstract_context × sac_abstract_state) ⇒ bool) ⇒ seL4_Word
⇒ instr"
  where
  cnd_jmp P n =
     (λs. (if (P s)
           then
             jmp n
           else return None) s)
```

Figure 4.6: Definitions of *jmp* and *cnd_jmp*.

4.1.3 Router Manager program structure

At the top level there are two parts to the functional specification Router Manager program. Firstly it waits for a message from the Control Network. Secondly it executes one of three branches of a switch on the received message.

The three branches are 'teardown Router', 'create Network A Router', and 'create Network B Router'. The first branch revokes all kernel objects which were created in setting up the Router thread, thereby destroying the Router and erasing anything the Router wrote to memory. The second branch also revokes all Router kernel objects, and then creates a new Router thread and gives it access to Network A. The third branch does the same as the second branch, except it gives access to Network B instead of Network A.

Once a branch has finished, the Router Manager program returns to waiting for a message from the Control Network. Figure 4.7 is illustrative to the structure.

At the stage that this part of the program runs, the Router Manager has just resumed from waiting on a message from its Control Network interface.

As covered in Section 2.4 and Subsection 4.1.3, the Router Manager's behaviour is quite simple.

Figure 4.7 shows precisely how the Router Manager algorithm is represented in the Router Manager functional specification. Also shown is the corresponding C code, which is very similar and should aid in understanding the flow of the unstructured code. .

C code

```
info = seL4_Wait(command_endpoint, NULL);
message = seL4_GetMR(0);
switch (message) {
  case 0:
    ⟨Teardown Router⟩
    break;

  case 1:
    ⟨Teardown Router⟩
    ⟨Setup Router (NIC A)⟩
    break;

  case 2:
    ⟨Teardown Router⟩
    ⟨Setup Router (NIC B)⟩
    break;

  default:
    ...
    break;
}
```

RM Line	**RM Instruction**
0	k_instr wait_for_message
1	u_instr get_message
2	cnd_jmp (λ(tc,rm). message rm = 0) 4,
3	jmp 13,
	⟨Teardown Router⟩
12	jmp 86
13	cnd_jmp (λ(tc, rm). message rm = 1) 15,
14	jmp 50
	⟨Teardown Router⟩
	⟨Setup Router (NIC A)⟩
49	jmp 86
50	cnd_jmp (λ(tc, rm). message rm = 2) 52,
51	jmp 86
	⟨Teardown Router⟩
	⟨Setup Router (NIC B)⟩

Figure 4.7: Comparison of C code and Functional specification Router Manager control flow

In Subsection 4.1.4 and Subsection 4.1.5 we will look at the two main tasks from the Router Manager shown above, 'Teardown Router' and 'Setup Router'. In the subsections following those, we will go into more detail examining some selected instructions from the two tasks.

4.1.4 Example: *Teardown Router*

As mentioned in Subsection 4.1.3, each of the branches of the Router Manager program switch will first teardown a currently established Router (if there is one). Figure 4.8 gives the Router Manager Functional specification representation of that teardown process. The process of tearing down the Router is identical in the three branches, however the line numbers and *jmp* and *cnd_jmp* targets (not branch conditions) will be different, here α and β.

The Router teardown process is one loop nested inside another, the outer loop iterates over the various untyped sizes and the inner loop iterates over the non-free untyped capabilities. At each

iteration of the inner loop a non-free untyped capability is freed by revoking it (revocation is discussed in Section 2.2 and Subsection 4.1.6). At each iteration of the outer loop, *all* non-free capabilities of the current size are freed.

In addition to destroying all of the kernel objects and capabilities associated with the old Router, we will sanitise all network interfaces. This is done to ensure that no residual information from communications with one instantiation of the Router can bleed into a future instantiation. Santising one interface on the present architecture consists of restarting it.

```
    u_instr (modify (λ(tc, rm).  (tc, rm⦇var_i:=0⦈)))
    jmp β
α:  cnd_jump (λ(tc, rm).  (free_items rm)!(var_i rm) ≥
      (total_items rm)!(var_i rm)) β
    k_instr revoke_ith_untyped
    u_instr (modify (λ(tc, rm).  (tc, free_items_update
      (λx.  x[(var_i rm) := (x !  (var_i rm)) + 1]) rm)))
    jmp α
β:  cnd_jmp (λ(tc, rm).  var_i rm < NUM_SIZES) α
    k_instr sanitise_network_cards
```

Figure 4.8: Router Manager Functional specification Router teardown.

The teardown process in the Router Manager Functional specification (see Subsection 4.1.4) mostly consists of revoking untyped objects which are kept in a two-dimensional array *untyped_items*, where the first dimension is untyped object size, and the second is just the numbering untypeds. Revoking all the untyped items therefore requires iterating over both the sizes of untypeds and the number of untypeds of each size. At each iteration of the loop a kernel call is performed- meaning that the kernel must run- so the loop cannot be done in one Router Manager Functional specification intruction.

The instruction executed at each iteration of the inner loop is *revoke_ith_untyped*, explained in Subsection 4.1.6. The *sac_abstract_state* variable *var_i* contains the iterator for untyped size and *free_items* is used as the iterator for the inner loop. *free_items* is an array containing the number of free (as in, unallocated) untyped items for each untyped size, so, for a given size, we are iterating over each non-free untyped of that size.

4.1.5 Example: *Setup Router*

In Subsection 4.1.3 we discussed how two of the three branches of the overall Router Manager program flow will set up a Router thread. Figure 4.9 illustrates how the Router Manager Functional specification performs the setup. In the two 'setup' branches, the line numbers and *jmp* and *cnd_jmp* targets (not the branch conditions) will be different, here they are α, β, γ, and δ.

Additionally, the outside network interface (either Network A or Network B) will vary between the two setup branches, here $\mathbb{I}$ is used in place of either 'A' or 'B'.

The process of setting up a new Router involves creating a new Thread Control Block for the Router thread and then populating it with a *CSpace*, virtual address space, and an *IPC Buffer* (see Section 2.2). Additionally we need to copy some capabilities into the Router's *CSpace*. There are two loops in the setup, the first maps a number of 'scratch' frames into the Router's virtual address space and the second copies a number of untyped capabilities into the Router's *CSpace*.

```
    k_instr create_router_cspace_object
    k_instr mutate_router_cspace_cap
    u_instr (modify (λ(tc, rm).  (tc, rm⦇var_i:=0⦈)))
    jmp β
α:  k_instr create_router_scratch_space
    k_instr map_router_scratch_frame
    k_instr move_router_scratch_frame
    u_instr (modify (λ(tc, rm).  (tc, rm⦇val_i:=val_i rm + 1⦈)))
β:  cnd_jmp (λ(tc, rm).  val_i rm < CLIENT_NUM_SCRATCH_FRAMES) α
    k_instr create_router_ipc_object
    k_instr map_router_ipc_frame
    k_instr create_router_tcb_object
    k_instr setup_router_tcb
    k_instr set_router_tcb_ip_sp
    k_instr copy_time_t_client
    u_instr (modify (λ(tc, rm).  (tc, rm⦇val_i:=0⦈)))
    jmp δ
γ:  k_instr copy_untyped_to_client
    u_instr (modify (λ(tc, rm).  (tc, rm⦇val_i:=val_i rm + 1⦈)))
δ:  cnd_jmp (λ(tc, rm).  val_i rm < CLIENT_NUM_UNTYPED_ITEMS) γ
    k_instr copy_asid_pool_to_client
    k_instr copy_nic_𝕀_to_client
    k_instr copy_nic_D_to_client
    k_instr copy_tcb_cap_to_client
    k_instr move_ipc_buffer_cap_to_client
    k_instr mutate_cspace_to_client
```

Figure 4.9: Router Manager Functional specification Router setup.

4.1.6 In-depth *instr* examples

The previous sections of this document have laid out the infrastructure for expressing the Router Manager program in the Functional specification as well as the intended structure and purpose of the Router Manager program. This sub-section aims to bring that altogether and to provide the

reader with some specific examples of code from the Router Manager program, how we wrote it and what it does in some detail. A line-by-line treatment would be very long so I have selected to present a few examples which should cover most 'sorts' of tasks in the program, and how they work.

The Router Manager Functional specification represents the Router Manager program as a list of instructions (unstructured code in the style of assembly), as discussed in Section 3.1. In this part of the document I will discuss several such 'instructions' from the Router Manager Functional specification's representation of the Router Manager program, *router_manager*. These particular examples also appear in Figure 4.8 and Figure 4.9. The full definition of *router_manager* can be found in Appendix A.

The very first thing that the Router Manager does, after booting, is to wait on the Control Network for a message (for more details see Section 2.4) which will contain either 'teardown Router', 'create Network A Router', and 'create Network B Router'. Here we will treat the process from the perspective of the Router Manager Functional specification. Figure 4.10 gives the definition in terms of a libseL4 interface function, *seL4_Wait*. The call must be lifted from *tc_monad* (see Subsection 4.2.1) to *rm_monad* since the libseL4 interface functions are defined across a generic thread state and not Router Manager specific state (see Subsection 3.2.2). *seL4_Wait* will be treated in depth in Subsection 4.2.4. The overall effect of this instruction is to set some registers up for the kernel, and then to trap into the kernel suspending itself until it receives a message.

```
definition
  wait_for_message ::
  (event option) rm_monad
  where
  wait_for_message =
   do command_endpoint' ← gets (λ(tc, rm). command_endpoint rm);
      lift_tc_rm (seL4_Wait command_endpoint' 0)
   od
```

Figure 4.10: Definition of *wait_for_message*

The main activity of *revoke_ith_untyped* is calling libseL4 interface function *seL4_CNode_Revoke*, via the Router Manager Functional specification abbreviation *revoke_untyped*, which lifts the libseL4 function and supplies some constant arguments.

revoke_untyped is given in Figure 4.12. Calling *seL4_CNode_Revoke* should set some registers up and trap into the kernel, so that the kernel can revoke the specified untyped object- destroying any kernel objects residing in the untyped object along with any capabilities refering to them. In the context of the whole program; when this is done on untyped memory objects which were used to set up a previous Router thread, it will result in stripping of the Router of its capabilities and then shut the Router thread itself down.

```
definition
  revoke_ith_untyped ::
  (event option) rm_monad
  where
  revoke_ith_untyped =
   do i ← gets (λ(tc, rm). var_i rm);
      free_items' ← gets (λ(tc, rm). free_items rm);
      untyped_items' ← gets (λ(tc, rm). untyped_items rm);
      revoke_untyped RM_SelfCSpace
        ((untyped_items'!i)!(unat (free_items'!i)))
   od
```

Figure 4.11: Definition of *revoke_ith_untyped*

```
definition revoke_untyped ::
  seL4_CPtr ⇒ seL4_CPtr ⇒ (event option) rm_monad
  where
  revoke_untyped dest_cspace dest_slot =
     lift_tc_rm (seL4_CNode_Revoke dest_cspace dest_slot
                                   (of_nat seL4_WordBits))
```

Figure 4.12: Definition of *revoke_untyped*.

The process of setting up a new Router is much more complex than tearing one down (see Subsection 4.1.5), so all of the tasks involved can't be broken down in this section. The example presented is the instruction that will create a kernel object for the Router's Thread Control Block (TCB) in the kernel. A TCB in the seL4 microkernel is quite close to what a reader will be familiar with from a Unix or other L4 TCB. Importantly it contains an Inter-Process Communication buffer (IPC buffer), the root CNode of the thread's CSpace (see Section 2.6), and the root of the thread's virtual address space. This important information will be populated in subsequent instructions- the create instruction should just provide us with a fresh TCB object, along with a capability to it.

```
definition
  create_router_tcb_object ::
  (event option) rm_monad
  where
  create_router_tcb_object =
   do item ← allocate_small_item;
      create_object item RM_SelfCSpace RM_Client_Tcb
                    seL4_TCBObject 0
   od
```

Figure 4.13: Definition of *create_router_tcb_object*.

As shown in Figure 4.13, we first call a function to 'allocate' a small kernel object (see Subsection 4.1.7) by updating some internal bookkeeping on untyped memory. Next, we call the libseL4 interface function *seL4_Untyped_Retype*, via Router Manager Functional specification abbreviation *create_object*. *create_object*, as shown in Figure 4.14, is merely a wrapper which lifts the libseL4 interface function *seL4_Untyped_Retypr* to *rm_monad*.

```
definition create_object ::
  seL4_CPtr ⇒ seL4_CPtr ⇒ seL4_CPtr ⇒ seL4_Word ⇒ seL4_Word
   ⇒ (event option) rm_monad
  where
  create_object untyped_item dest_cnode dest_slot targettype targetsize =
     lift_tc_rm (seL4_Untyped_Retype untyped_item targettype targetsize
                                     dest_cnode 0 0 dest_slot 1)
```

Figure 4.14: Definition of *create_object*.

4.1.7 Internal helper function examples

For the most part we tried to mirror the internal structure of the Router Manager C implementation for the Functional specification program. At some points this meant clumping user transitions together with a kernel call as shown in Subsection 4.1.6. When this happened, there were often internal helper functions in the C implementation so instead of 'inlining' these purely internal transitions they were represented as *rm_monad* computations to be used in other instructions.

get_message, shown in Figure 4.15, calls libseL4 helper function *seL4_GetMR* to retrieve some information from a message register. Message register being a user view abstraction of the Router Manager thread's IPC buffer. The structure of libseL4 and the functions it exposes is discussed in much more detail in section Section 4.2. In this case the message refered to is from the Control Network, and contains direction on what to do next.

```
definition
  get_message ::
  (event option) rm_monad
  where
  get_message =
   do m ← gets (λ(tc, rm). seL4_GetMR 0 tc);
      modify (λ(tc, rm). (tc, rm(|message := m|)));
      return None
   od
```

Figure 4.15: Definition of *get_message*.

allocate_small_item is shown in Figure 4.16. This helper function is called to update bookkeeping information on untyped memory (see Subsection 4.1.6 for an example of its use). It is an abbreviation for *allocate_item*.

```
definition allocate_small_item ::
  seL4_CPtr rm_monad
  where
  allocate_small_item = allocate_item SIZE_4K
```

Figure 4.16: Definition of *allocate_small_item*.

Figure 4.17 gives the definition for *allocate_item*. First, we decrease the number of 'free' untyped items of the given size in *free_items*. Second we find the next 'free' untyped item in *untyped_items* and return a pointer to it.

```
definition
  allocate_item ::
  untyped_size ⇒ seL4_CPtr rm_monad
  where
  allocate_item sz =
     do modify (λ(tc, rm). (tc, free_items_update (λx. x[sz := (x!sz) - 1]) rm));
        ptr ← gets (λ(tc, rm).
                        ((untyped_items rm)!sz)!(unat ((free_items rm)!sz)));
        return ptr
     od
```

Figure 4.17: Definition of *allocate_item*.

4.2 libseL4 specification

We have now defined the foundations of the Router Manager Functional specification, and shown some examples of how the Router Manager component is expressed up to where the kernel is called. Beyond that, there are libseL4 functions setting up and trapping into the kernel. To completely specify the Router Manager we need to specify the subset of libseL4 that the Router Manager uses.

Just as the Router Manager instructions are treated as indivisible monadic computations, so are the libseL4 functions. In the future there may be some reason to implement pre-emptible libseL4 interface functions, however it seems like a reasonable assumption that seL4 user systems (especially those that you would like to verify!) can separately be proven to have interference freedom from the rest of the system, with regards to their *abstract_thread_context* (see Subsection 4.2.2). If the *abstract_thread_context* is interference free then there is no harm in representing the libseL4

interface functions as atomic.

As with the previous section, it would be infeasible to present the entire specification of the libseL4 library here. The foundational definitions are given in this section along with examples which are meant to be illustrative of how the specification works.

4.2.1 The *tc_monad*

```
types 'a tc_monad = (thread_abstract_context, 'a) nondet_monad
```

Figure 4.18: Definition of *tc_monad*.

As the libseL4 interface functions are meant to be user program generic, they are defined across a thread state rather than the Router Manager, or Secure Access Controller, state. These computations are represented by the *tc_monad*, illustrated in Figure 4.18. Similarly to the Router Manager instructions, functions which trap into the kernel are expected to return some *SyscallEvent sy* of the appropriate sort, where the C implementation of that same function would have set up a software interrupt.

4.2.2 The *thread_abstract_context*

```
types thread_abstract_context = seL4_IPCBuffer × user_context
```

Figure 4.19: Definition of *thread_abstract_context*.

thread_abstract_context represents user state which is common to all seL4 user programs, and which is also necessary for the libseL4 library functions. Shown in Figure 4.19, the *seL4_IPCBuffer* component of the state is the Inter-Process Communication Buffer (IPC Buffer) of the user program thread and the *user_context* represents the user view of machine registers. The IPC Buffer is a buffer for messages sent to the user thread by the kernel or other processes. The *user_context* type comes from the seL4 kernel Abstract specification. Defining the libseL4 functions in terms of *user_context* is convenient because we wish to use the seL4 kernel Abstract specification in our implementation of the overall system process.

4.2.3 libseL4 specification structure

The seL4 microkernel provides six syscalls and several dozen capability invocations. Capability invocations are particular instances of syscalls: the user gives the kernel some capability while making a syscall and the kernel interprets that as some *invocation* on the given capability. For

example, *seL4_Wait* is a syscall and *seL4_CNode_Revoke* is a capability invocation (see Subsection 4.1.6). The libseL4 library provides C functions that will set up both the syscalls and the capability invocations. Because the capability invocations depend on the syscalls, we can view the libseL4 syscall setup functions as the libseL4 interface to the kernel and the capability invocations as the user's interface to libseL4. Because of this, in the libseL4 specification the syscalls are part of the *seL4 interface* while the capability invocations are part of the *libseL4 interface*.

The C implementation of the libseL4 library can be broken down conceptually into a seL4 interface, a libseL4 interface, and a set of types which are used to represent the user view of kernel objects as well as to pass data into the kernel.

We implemented the libseL4 specification as a package of types that user programs can use to represent seL4 kernel objects, on top of which are functions to set and get data from the *thread_abstract_context* (see Subsection 4.2.5), seL4 interface functions (syscalls), and then libseL4 interface functions.

Only the parts of the libseL4 interface used by the Router Manager program have been implemented, however what has been implemented is generic enough that other user programs could easily use the same libseL4 specification, assuming architecture compatibility. Filling in the rest of the interface would be straight-forward and not time-consuming, since everything that the functions depend on has been set up in the process of getting a subset working.

4.2.4 In-depth libseL4 examples

Presented here are some examples of libseL4 and seL4 interface functions. Their purpose may not be immediately obvious from the definitions: in each case some information is marshalled and loaded into registers where the kernel can find it and special values are loaded into other registers so that the kernel can determine the user's intent. Then a software interrupt is generated, trapping into the kernel so that the desired operation can be performed.

Careful readers will notice that the libseL4 interface functions load data into message registers which the seL4 interface functions then put into machine registers. This is definitely overly complex and costly to performance but exists in the current implementation for historical reasons, and because code is automatically generated. Correcting this behaviour is very much outside of the scope of this thesis project, however it was noticed and modifying these definitions when or if this behaviour is changed would not be burdensome (indeed it would make the definitions smaller).

seL4_Call, shown in Figure 4.20, is part of what we call the seL4 interface of libseL4, it provides a convenient way to generate a seL4 syscall. We take some arguments and load them into registers, then move some data from message registers into machine registers (or at least the user's view

of the machine registers).

```
definition seL4_Call ::
  seL4_CPtr ⇒ seL4_MessageInfo ⇒ (event option) tc_monad
  where
  seL4_Call dest msgInfo =
     do modify (λ(i,u). (i, u (R0 := dest)));
        modify (λ(i,u). (i, u (R1 := msgInfo)));
        MR0 ← gets (seL4_GetMR 0);
        modify (λ(i,u). (i, u (R2 := MR0)));
        MR1 ← gets (seL4_GetMR 1);
        modify (λ(i,u). (i, u (R3 := MR1)));
        MR2 ← gets (seL4_GetMR 2);
        modify (λ(i,u). (i, u (R4 := MR2)));
        MR3 ← gets (seL4_GetMR 3);
        modify (λ(i,u). (i, u (R5 := MR3)));
        modify (λ(i,u). (i, u (R7 := seL4_SysCall)));
        return (Some (SyscallEvent SysCall))
     od
```

Figure 4.20: Definition of *seL4_Call*.

Figure 4.21 presents *seL4_Wait*, which is also a seL4 interface function. It sets up a seL4 syscall similarly to *seL4_Wait*. It is often used directly by user programs, for example the Router Manager (see Figure 4.21).

```
definition seL4_Wait ::
  seL4_CPtr ⇒ seL4_Word ⇒ (event option) tc_monad
  where
  seL4_Wait src sender =
     do modify (λ(i,u). (i, u (R0 := src)));
        modify (λ(i,u). (i, u (R7 := seL4_SysWait)));
        return (Some (SyscallEvent SysWait))
     od
```

Figure 4.21: Definition of *seL4_Wait*.

seL4_CNode_Revoke is an example of a capability invocation or *libseL4 interface* function. Shown in Figure 4.22, its purpose is to revoke any children of the invoked capability. In the case of a capability to an untyped memory object, any kernel object created in that untyped will be a child and so revoking the untyped capability will destroy any objects created inside. The Router Manager program uses this function in tearing down the Router thread, by repeatedly revoking untyped capabilities (see Figure 4.8).

The purpose of *seL4_Untyped_Retype*, shown in Figure 4.23 is to create a new kernel object in the memory covered by invoked untyped capability. In practice it places some data into message

```
definition
  seL4_CNode_Revoke ::
  seL4_CNode ⇒ seL4_Word ⇒ 8 word
   ⇒ (event option) tc_monad
  where
  seL4_CNode_Revoke service index depth =
     do seL4_SetMR 0 index;
        seL4_SetMR 1 (of_nat (unat depth) && 0xff);
        seL4_Call service (seL4_MessageInfo 2 0 (of_nat (unat CNodeRevoke)) 0)
     od
```

Figure 4.22: Definition of *seL4_CNode_Revoke*.

registers and calls *seL4_Call*, which will do more setup and generate a software interrupt. This function is used extensively in the Router Manager's setup of a Router thread (see Figure 4.9). *seL4_Untyped_Retype* is also a *libseL4 interface* function.

```
definition
  seL4_Untyped_Retype ::
  seL4_CPtr ⇒ seL4_Word ⇒ seL4_Word ⇒ seL4_CPtr ⇒ seL4_Word ⇒ seL4_Word
   ⇒ seL4_CPtr ⇒ seL4_Word
   ⇒ (event option) tc_monad
  where
  seL4_Untyped_Retype service type size_bits root node_index node_depth
                      node_offset node_window =
     do seL4_SetCap 0 root;
        seL4_SetMR 1 size_bits;
        seL4_SetMR 2 node_index;
        seL4_SetMR 3 node_depth;
        seL4_SetMR 4 node_offset;
        seL4_SetMR 5 node_window;
        seL4_Call service (seL4_MessageInfo 6 1 (of_nat (unat UntypedRetype)) 0)
     od
```

Figure 4.23: Definition of *seL4_Untyped_Retype*.

4.2.5 libseL4 helper function examples

The functions presented in this sub-section are used heavily in the seL4 and libseL4 interface functions of the libseL4 specification. They are quite straight-forward but are presented so that the continuing examples from Subsection 4.1.6 to Subsection 4.2.4 are completely described (the complete definitions of all other parts of the specifications are given in Appendix A, Appendix B, and Appendix C).

The function *seL4_GetMR* is heavily used in the definition of *seL4_Call*, is shown in Figure 4.24. It simply returns the value inside some message register in the *thread_abstract_context* of the user program thread. It does not return any event because it is used inside the 'atomic' libseL4 and seL4 interface functions.

```
definition seL4_GetMR ::
  nat ⇒ thread_abstract_context ⇒ seL4_Word
  where
  seL4_GetMR n = (λ(i, u). (ipc_msg i) ! n)
```

Figure 4.24: Definition of *seL4_GetMR*.

The function *seL4_SetMR*, shown in Figure 4.25, will set the given message register to the given value. It does not return any event because it too is used inside of other 'atomic' functions.

```
definition seL4_SetMR ::
  nat ⇒ seL4_Word ⇒ (event option) tc_monad
  where
  seL4_SetMR n w =
     do modify (λ(i, u). ((ipc_msg_update (λx. x[n := w]) i, u)));
        return None
     od
```

Figure 4.25: Definition of *seL4_SetMR*.

4.2.6 Lifting from *tc_monad* to *rm_monad*

The aim of the libseL4 specification is to have a generic interface that any user program specification can use. The libseL4 functions, however, are defined in terms of the *tc_monad*, and it is unlikely that user program specifications could be fully captured by *tc_monad*.

The Router Manager functional specification, for example, uses *sac_abstract_state* in its *rm_monad*. Therefore in order for Router Manager program specification to use the libseL4 functions we need a way to lift functions from the *tc_monad* to *rm_monad*.

The function *lift_tc_rm*, shown in Figure 4.26, lifts a computation from the *tc_monad* to the *rm_monad*. It carries the *tc_monad* computation on the *thread_abstract_context* part of the Router Manager state, and then recombines the resulting state with the rest of the Router Manager state and returns the return value of the *tc_monad* computation.

```
definition
  lift_tc_rm ::
  'a tc_monad ⇒ 'a rm_monad
   where
  lift_tc_rm f =
     (λ(tc, rm). ((λ(e, tc'). (e, (tc', rm))) ‘ fst (f tc), snd (f tc)))
```

Figure 4.26: Definition of *lift_tc_rm*.

4.3 System process

As mentioned in Chapter 1, showing that the Router Manager Functional specification is a refinement of the Router Manager Security model will require some way to create a process from each. We did this by defining a generic system automaton in terms of a user specification and a kernel specification. The idea of the automaton is to capture all of the potential executions of the user and kernel with arbitrary scheduling, of course covering all user to kernel transitions and kernel to user transitions.

Aside from the main goal of this thesis project- to formally specify the Router Manager- we also implemented the generic system automaton to allow any system processes to be defined and we instantiated it with a system process for the Router Manager Functional specification. This section presents the generic system automaton, the system process as it is defined for the Router Manager Functional specification, how the kernel specification is defined, and how the user specification is defined.

4.3.1 The *global_automaton*

In this sub-section we will introduce some necessary types and then the definition of the *global_automaton* itself.

The *mode* datatype, depicted in Figure 4.27, represents our notion of which mode the entire system is running in at a given time. At all times the kernel is considered to be in one of: *Idle mode*, *Kernel mode*, or *User mode*. The distinction is rather self-explanatory, the system is executing in *Idle mode* when neither a thread nor the kernel is running, *Kernel mode* when the kernel is running (it cannot be interrupted by the scheduler or pre-empted), and *User mode* when some user thread is running. This is illustrated in Figure 3.3.

```
datatype mode = UserMode | KernelMode | IdleMode
```

Figure 4.27: Definition of *mode*.

As shown in Figure 4.28, *user_mem* represents a user view of system memory. 32 bit addresses

potentially map to data in memory, dependant on whether the current user thread should be able to access that address or not. Along with *user_context* and the abstract *Instruction pointer*, this makes up the user view of the machine.

```
types user_mem = word32 ⇒ word8 option
```

Figure 4.28: Definition of *user_mem*.

The type *user_context* is meant to capture the user program's view of the system's registers. As can be seen in Figure 4.29, the type is a map from *register* to *data*. The types of *register* and *data* are not given here for the sake of length, however the former is just an abstract type for a register's name (for example *R0*) and the latter is a 32 bit word.

```
types user_context = register ⇒ data
```

Figure 4.29: Definition of *user_context*.

The *global_automaton* is intended to, given a user specification and kernel specification, return all of the system's transitions. For consistency with the refinement machinery from the seL4 refinement, the transitions are broken up structurally into the different *global_transition* types. A *global_transition* is either a *KernelTransition*, *UserTransition*, *UserEventTransition*, or an *IdleEvent-Transition*, however the *UserEventTransition* is no longer used because of the user having its own specification in this definition.

As shown in Figure 4.30 the definition is broken up into different types of transitions, each giving the set of possible system transitions. For example, the kernel transitions consist of the union of some transitions from *KernelMode* to *UserMode* and from *KernelMode* to *IdleMode*. This is because the kernel is expected to not generate kernel events and hence once the system enters *KernelMode* it can only transition to *UserMode* or *IdleMode* (presumably if there are no user threads ready to run).

For the definition of 'allowed' kernel transitions, *global_automaton* appeals to the given kernel specification *kernel_call*, and for 'allowed' user transitions it appeals to the user specification *user_transition*. Additionally we allow the user threads to generate an *Interrupt* event at any time, since these will not be explicitly returned by the user specification. Finally, the only 'allowed' *IdleMode* transition is to *KernelMode* via an *Interrupt* event.

The type of *global_automaton* is not concrete, there is a type variable $'k$ which should be instantiated with a seL4 kernel state type, for example the seL4 Abstract specification *state*. The types of *user_transition* and *kernel_call* also depend on $'k$, so they must use the same kernel state. Fully applied, *global_automaton* is a relation on $'k$ *global_state*, representing the possible transitions between $'k$ *global_state* states.

```
definition
  global_automaton ::
    ((user_context × word32 × 'k) × (user_context × word32 × 'k)
      × mode × event option) set
     ⇒ (event ⇒ ((user_context × 'k) × mode × (user_context × 'k)) set)
     ⇒ global_transition ⇒ ('k global_state × 'k global_state) set
  where
  global_automaton user_transition kernel_call t ≡ case t of
    KernelTransition ⇒
       { ( ((uc, ip, s), KernelMode, Some e),
           ((uc', uc' LR_svc, s'), UserMode, None) ) |uc ip s uc' s' e.
             ((uc, s), UserMode, (uc', s')) ∈ kernel_call e }
     ∪ { ( ((uc, ip, s), KernelMode, Some e),
           ((uc', ip', s'), IdleMode, None) ) |uc ip s uc' ip' s' e.
             ((uc, s), IdleMode, (uc', s')) ∈ kernel_call e }
  | UserTransition ⇒
       { ( (g, UserMode, None), (g', UserMode, e) ) |g g' e.
       (g, g', UserMode, e) ∈ user_transition}
     ∪ { ( ((uc, ip, g), UserMode, None),
           ((uc'(FaultInstruction := ip'), ip', g'), KernelMode, Some ev) )
           |uc ip g uc' ip' g' ev.
             ((uc, ip, g), (uc', ip', g'), KernelMode, Some ev) ∈
               user_transition}
     ∪ { ( (g, UserMode, None), (g, KernelMode, Some Interrupt) ) |g. True }
  | IdleEventTransition ⇒
     { ( (g, IdleMode, None), (g, KernelMode, Some Interrupt) ) |g. True }
```

Figure 4.30: Definition of *global_automaton*.

4.3.2 The *ADT_I*

Given the generic definitions that we implemented, as presented in Subsection 4.3.1, we may now define the entire system process for the seL4 microkernel and the Router Manager program. Shown in Figure 4.31, the system process is called *ADT_I*. *ADT_I* is a record containing three functions: *Init*, *Fin*, and *Step*.

Init is a function which initializes the system process by setting up a *state global_state* which corresponds to a given *observable* state (see Figure 2.2). For any system process, we must choose some kernel view of the system, this will be the state of the kernel specification. *observable* state is a representation of the system state which is not dependant on that kernel view of the system. Because it is not dependant on any internal kernel view of the system, the *observable* state is comparable between system processes which could have different kernel specifications. As mentioned in Subsection 4.3.1, the *global_automaton* is a relation over some *'k global_state*, which in the case of the *ADT_I* becomes concrete type *state global_state*.

The function *Fin* does more or less the reverse of *Init*, it finalizes the system process. *Fin* will give the *observable* state which corresponds to a given *state global_state*. In forward simulation, for a particular system process, we will use *Init* and *Fin* to encode an arbitrary state into that process's internal representation. Then after taking a step, we decode the resulting states into something comparable with another system process.

The *Init* and *Fin* functions given in *ADT_I*, *Fin_A* and *Init_A*, are both taken from the seL4 Abstract specification automaton that our definition has been adapted from, since *ADT_I* uses the Abstract kernel specification.

The third field, *Step*, is probably the most interesting. *Step* encodes all of the valid transitions of the system process. For each *global_transition*, it gives a set of all system transitions of that type. For *ADT_I*, *Step* is *global_automaton* instantiated in terms of *do_user_op*, *uop_I*, and *kernel_call_A*.

The function *uop_I* is the unlifted user specification and is discussed in Subsection 4.3.4. *do_user_op* is a helper definition from the generic automaton machinery I adapted, it lifts *uop_I* to transitions on *state global_state*. *kernel_call_A* is the kernel specification for the seL4 Abstract specification, and is discussed in Subsection 4.3.3.

The type of *ADT_I* uses abbreviation *data_type* for historical reasons. For type variables *(α, β, γ)*, *data_type* will be a record with fields *Init* :: $\beta \Rightarrow \alpha$, *Fin* :: $\alpha \Rightarrow \beta$, and *Step* :: $\gamma \Rightarrow (\alpha \times \alpha)$.

```
definition
  ADT_I :: (state global_state, observable, global_transition) data_type
where
 ADT_I ≡
  (| Init = λs. Init_A,
    Fin = Fin_A,
    Step = global_automaton (do_user_op uop_I) kernel_call_A|)
```

Figure 4.31: Definition of *ADT_I*.

4.3.3 Kernel specification

The definition of *kernel_call_A* is given here, in Figure 4.32, for the curious. It gives the type of our seL4 Abstract kernel specification. The definition obviously does not define the behaviour of the kernel directly- there are many thousands of lines of definition and proof to it.

4.3.4 User specification

The user specification for our system process in general just needs to specify the valid *user_state* transitions and which sort of *event* they generate. We say *user_state* transitions rather than *state*

```
definition
  kernel_call_A ::
  event ⇒ ((user_context × state) × mode × (user_context × state)) set
  where
  kernel_call_A e ≡
      {(s, m, s'). s' ∈ fst (split (kernel_entry e) s) ∧
                   m = (if ct_running (snd s') then UserMode else IdleMode)}
```

Figure 4.32: Definition of *kernel_call_A*.

global_state transitions, since there is a generic function, *do_user_op*, to lift to *state global_state*.

In the particular case of the Router Manager Functional specification, we have already examined the *sac_abstract_state* and the *rm_monad* in Section 4.1. What is presented in this subsection is how we lift *rm_monad* computations to *user_state* transitions and how we lift *user_state* transitions to *state global_state* transitions.

uop_I, as shown in Figure 4.33, decodes the program text of the currently running thread from user memory giving a list of *user_state* transitions. It then decodes the program counter from the user state and 'executes' that instruction from the list, at the current user state, returning the result.

```
definition
  uop_I ::
  user_state ⇒ (user_state × event option) set
  where
  uop_I us =
    ((decode_program_text
      (project_program_text us))!(project_program_counter us)) us
```

Figure 4.33: Definition of *uop_I*.

decode_program_text maps a region of data, containing machine instructions, to a list of *user_state* instructions. In practice the result will be an arbitrary list for *untrusted* processes or the *router_manager* program, lifted to *user_state*, for the *trusted* Router Manager process. Therefore when reasoning about the system process for the Router Manager Functional specification, transitions when an *unstrusted* process is running can be anything and transitions when the Router Manager process is running are those from Functional specification.

project_program_text projects the region of data, containing the currently running process's program text, out of the user process's entire view of memory (contained in *user_state*).

project_program_counter projects the program counter out of the *user_state*, which will be the abstract *instruction_pointer*.

Chapter 5

Conclusions

The goal of this project was to formally specify the functionality of the Router Manager component of the Secure Access Controller, so that it could serve as a midway point in the refinement from the SAC Security model to its C implementation. Since the C implementation uses functions from the common seL4 user library libseL4, it was necessary along the way to specify the behaviour of a libseL4 subset as well.

5.1 Contributions

We succeeded in implementing the Router Manager Functional specification, leaving the specification of libseL4 as a separate object. Additionally, we implemented the subset of libseL4 needed for the Router Manager program as well as the framework for showing refinement. The libseL4 subset was designed to be general enough that another seL4 user program could use its functions, by writing a simple lift from the libseL4 state to the user program's state. The refinement framework was designed so that, similarly, it could be used to express refinement between other seL4 user program specifications.

For the above reasons, this project has been quite helpful as a case study in seL4 user program verification.

5.2 Future Work

The next steps after this thesis will be to define a system process for the SAC Security model Router Manager program and then to prove that the Router Manager Functional specification

refines it. This will most likely necessitate modifying the SAC Security model. Security model transitions need to be changed so that they can be lifted to *user_state* transitions which return an *event option*.

Further into the future, we will need to show that the C implementation of the Router Manager refines the Router Manager Functional specification. Doing so will likely require adapting the system process definitions a little. Because where the system process for the Functional specification takes the seL4 Abstract specification, the system process for the Router Manager's C implementation should take a different kernel specification.

Appendix A

Router Manager Functional Specification

theory *RouterManager_I*

imports *libseL4Interface*

begin

A.1 The Router Manager Functional specification program

Router Manager constant definitions

definition *CLIENT_CPTR_TCB :: "nat"* **where** *"CLIENT_CPTR_TCB = 1"*
definition *CLIENT_CPTR_CSPACE :: "nat"* **where** *"CLIENT_CPTR_CSPACE = 2"*
definition *CLIENT_CPTR_IPC_BUFFER :: "nat"* **where** *"CLIENT_CPTR_IPC_BUFFER = 4"*
definition *CLIENT_CPTR_ASID_POOL :: "nat"* **where** *"CLIENT_CPTR_ASID_POOL = 6"*
definition *CLIENT_CPTR_IO_PORT :: "nat"* **where** *"CLIENT_CPTR_IO_PORT = 7"*
definition *CLIENT_CPTR_TIMER_ENDPOINT :: "nat"* **where** *"CLIENT_CPTR_TIMER_ENDPOINT = 8"*
definition *CLIENT_CPTR_FIRST_UNTYPED_ITEM :: "nat"* **where**
"CLIENT_CPTR_FIRST_UNTYPED_ITEM = 1024"
definition *CLIENT_CPTR_FIRST_SCRATCH_ITEM :: "nat"* **where**
"CLIENT_CPTR_FIRST_SCRATCH_ITEM = 2048"

definition *UNTYPED_ITEM_SIZE_BITS :: "nat"* **where** *"UNTYPED_ITEM_SIZE_BITS = 20"*

definition *CLIENT_NUM_SCRATCH_FRAMES :: "nat"* **where** *"CLIENT_NUM_SCRATCH_FRAMES = 32"*
definition *CLIENT_NUM_UNTYPED_ITEMS :: "nat"* **where** *"CLIENT_NUM_UNTYPED_ITEMS = 100"*
definition *CLIENT_CSPACE_SIZE_BITS :: "nat"* **where**
"CLIENT_CSPACE_SIZE_BITS = (UNTYPED_ITEM_SIZE_BITS - seL4_SlotBits)"
definition *CLIENT_IPC_BUFFER_VADDR :: "nat"* **where**
"CLIENT_IPC_BUFFER_VADDR = 0xd0002000"
definition *CLIENT_SCRATCH_VADDR :: "nat"* **where** *"CLIENT_SCRATCH_VADDR = 0xd0003000"*
definition *CLIENT_ENTRY_POINT :: "nat"* **where** *"CLIENT_ENTRY_POINT = 0x00010000"*
definition *CLIENT_PRIORITY :: "nat"* **where** *"CLIENT_PRIORITY = 128"*

definition *RM_Null :: "seL4_Word"* **where** *"RM_Null = 0"*
definition *RM_SelfCSpace :: "seL4_Word"* **where** *"RM_SelfCSpace = 2"*
definition *RM_AsidPool :: "seL4_Word"* **where** *"RM_AsidPool = 6"*
definition *RM_IOPort :: "seL4_Word"* **where** *"RM_IOPort = 7"*

definition *RM_Client_Tcb :: "seL4_Word"* **where** *"RM_Client_Tcb = 1000"*
definition *RM_Client_CSpace :: "seL4_Word"* **where** *"RM_Client_CSpace = 1001"*
definition *RM_Client_IpcBuffer :: "seL4_Word"* **where** *"RM_Client_IpcBuffer = 1002"*
definition *RM_TempSlot :: "seL4_Word"* **where** *"RM_TempSlot = 1003"*

types *untyped_size = nat*
definition *SIZE_4K :: "nat"* **where** *"SIZE_4K = 0"*
definition *SIZE_1MB :: "nat"* **where** *"SIZE_1MB = 1"*
definition *NUM_SIZES :: "nat"* **where** *"NUM_SIZES = 2"*

A.2 The sac_abstract_state

record *sac_abstract_state =*

ip :: seL4_Word

```
  untyped_items :: "seL4_CPtr list list"
  free_items :: "seL4_Word list"
  total_items :: "seL4_Word list"

  info :: seL4_MessageInfo
  message :: seL4_Word
  var_i :: nat

  RM_RWGC_NIC_A_cap :: seL4_CPtr
  RM_RWGC_NIC_B_cap :: seL4_CPtr
  RM_RWGC_NIC_D_cap :: seL4_CPtr

  client_vspace :: seL4_CPtr
  virtualirq_endpoint :: seL4_CPtr
  command_endpoint :: seL4_CPtr
```

A.3 The rm_monad and Router Manager execution

```
types 'a rm_monad = "((thread_abstract_context × sac_abstract_state), 'a) nondet_monad"
types instr = "(event option) rm_monad"
types program = "instr list"

definition
  lift_tc_rm ::
  "'a tc_monad ⇒ 'a rm_monad"
   where
  "lift_tc_rm f =
     (λ(tc, rm). ((λ(e, tc'). (e, (tc', rm))) ‘ fst (f tc), snd (f tc)))"


definition incr_pc ::
  "unit rm_monad"
  where
  "incr_pc = modify (λ(tc, rm). (tc, ip_update (λip. ip + 1) rm))"

definition k_instr ::
  "instr ⇒ instr"
  where
  "k_instr = id"
```

```
definition u_instr ::
  "'a rm_monad ⇒ instr"
  where
  "u_instr m =
     do m;
        incr_pc;
        return None
     od"

definition jmp ::
  " seL4_Word ⇒ instr"
  where
  "jmp n =
     do modify (λ(tc, rm). (tc, ip_update (λx. x + 1) rm));
        return None
     od"

definition cnd_jmp ::
  "((thread_abstract_context × sac_abstract_state) ⇒ bool) ⇒ seL4_Word ⇒ instr"
  where
  "cnd_jmp P n =
     (λs. (if (P s)
           then
             jmp n
           else return None) s)"
```

A.4 Router Manager helper functions

```
definition
  allocate_item ::
  "untyped_size ⇒ seL4_CPtr rm_monad"
  where
  "allocate_item sz =
     do modify (λ(tc, rm). (tc, free_items_update (λx. x[sz := (x!sz) - 1]) rm));
        ptr ← gets (λ(tc, rm). ((untyped_items rm)!sz)!(unat ((free_items rm)!sz)));
        return ptr
     od"

definition allocate_large_item ::
  "seL4_CPtr rm_monad"
```

where
"allocate_large_item = allocate_item SIZE_1MB"

definition *allocate_small_item ::*
"seL4_CPtr rm_monad"
where
"allocate_small_item = allocate_item SIZE_4K"

definition
get_message ::
"unit rm_monad"
where
"get_message =
do m ← gets (λ(tc, rm). seL4_GetMR 0 tc);
modify (λ(tc, rm). (tc, rm(|message := m|)));
return ()
od"

A.5 Router Manager instructions

definition *get_cspace_capdata ::*
"seL4_Word ⇒ seL4_CapData"
where
"get_cspace_capdata cspace_size_bits =
seL4_CapData (of_nat (seL4_WordBits - unat cspace_size_bits)) 0"

definition *try_copy_cap ::*
"seL4_CPtr ⇒ seL4_CPtr ⇒ seL4_CPtr ⇒ seL4_CPtr ⇒ seL4_CapRights
⇒ (event option) rm_monad"
where
"try_copy_cap dest_cspace dest_slot src_cspace src_slot rights =
lift_tc_rm (seL4_CNode_Copy dest_cspace dest_slot (of_nat seL4_WordBits)
src_cspace src_slot (of_nat seL4_WordBits) rights)"

definition *copy_cap ::*
"seL4_CPtr ⇒ seL4_CPtr ⇒ seL4_CPtr ⇒ seL4_CPtr ⇒ seL4_CapRights
⇒ (event option) rm_monad"
where
"copy_cap dest_cspace dest_slot src_cspace src_slot rights =
try_copy_cap dest_cspace dest_slot src_cspace src_slot rights"

```
definition create_object ::
  "seL4_CPtr ⇒ seL4_CPtr ⇒ seL4_CPtr ⇒ seL4_Word ⇒ seL4_Word
   ⇒ (event option) rm_monad"
  where
  "create_object untyped_item dest_cnode dest_slot targettype targetsize =
     lift_tc_rm (seL4_Untyped_Retype untyped_item targettype targetsize dest_cnode
                 0 0 dest_slot 1)"

definition map_frame ::
  "seL4_CPtr ⇒ seL4_CPtr ⇒ seL4_Word ⇒ seL4_CapRights
   ⇒ (event option) rm_monad"
  where
  "map_frame vspace frame vaddr perms =
     lift_tc_rm (seL4_ARM_Page_Map frame vspace vaddr
                 perms seL4_ARM_Default_VMAttributes)"

definition move_cap ::
  "seL4_CPtr ⇒ seL4_Word ⇒ seL4_CPtr ⇒ seL4_CPtr ⇒ (event option) rm_monad"
  where
  "move_cap dest_cspace dest_slot src_cspace src_slot =
     lift_tc_rm (seL4_CNode_Move dest_cspace dest_slot (of_nat seL4_WordBits)
                              src_cspace src_slot (of_nat seL4_WordBits))"

definition revoke_untyped ::
  "seL4_CPtr ⇒ seL4_CPtr ⇒ (event option) rm_monad"
  where
  "revoke_untyped dest_cspace dest_slot =
     lift_tc_rm (seL4_CNode_Revoke dest_cspace dest_slot
                                   (of_nat seL4_WordBits))"

definition mutate_cap ::
  "seL4_CPtr ⇒ seL4_Word ⇒ seL4_CPtr ⇒ seL4_Word ⇒ seL4_CapData
   ⇒ (event option) rm_monad"
  where
  "mutate_cap dest_cspace dest_slot src_cspace src_slot data =
     lift_tc_rm (seL4_CNode_Mutate dest_cspace dest_slot
                                   (of_nat seL4_WordBits) src_cspace  src_slot
                                   (of_nat seL4_WordBits) data)"
```

definition

```
setup_tcb ::
"seL4_CPtr ⇒ seL4_CPtr ⇒ seL4_CapData ⇒ seL4_CPtr ⇒ seL4_Word ⇒ seL4_Word
 ⇒ seL4_CPtr
 ⇒ (event option) rm_monad"
```

where

```
"setup_tcb tcb cspace cspace_data vspace priority buffer_vaddr buffer_frame =
   lift_tc_rm (seL4_TCB_Configure tcb 0 (of_nat (unat priority)) cspace
                                  cspace_data vspace 0
                                  buffer_vaddr buffer_frame)"
```

definition

```
set_tcb_ip_sp ::
"seL4_CPtr ⇒ seL4_Word ⇒ seL4_Word ⇒ (event option) rm_monad"
```

where

```
"set_tcb_ip_sp tcb pc sp =
   lift_tc_rm (seL4_TCB_WriteRegisters tcb True 0 2
                                       (|upc=pc, usp=sp, ucpsr=0, ur0=0, ur1=0, ur8=0
                                        , ur9=0, ur10=0, ur11=0, ur12=0, ur4=0,
                                        ur2=0, ur3=0, ur5=0, ur6=0, ur7=0, ur14=0|))"
```

definition

```
sanitise_network_cards ::
"(event option) rm_monad"
```

where

```
"sanitise_network_cards = undefined"
```

definition

```
copy_nic_A_to_client ::
"(event option) rm_monad"
```

where

```
"copy_nic_A_to_client =
 do cap ← gets (λ(tc, rm). RM_RWGC_NIC_A_cap rm);
    try_copy_cap RM_Client_CSpace cap RM_SelfCSpace
                 cap seL4_AllRights
 od"
```

definition

```
copy_nic_B_to_client ::
```

```
  "(event option) rm_monad"
  where
  "copy_nic_B_to_client =
   do cap ← gets (λ(tc, rm). RM_RWGC_NIC_B_cap rm);
      try_copy_cap RM_Client_CSpace cap RM_SelfCSpace
                   cap seL4_AllRights
   od"

definition
  copy_nic_D_to_client ::
  "(event option) rm_monad"
  where
  "copy_nic_D_to_client =
   do cap ← gets (λ(tc, rm). RM_RWGC_NIC_D_cap rm);
      try_copy_cap RM_Client_CSpace cap RM_SelfCSpace
                   cap seL4_AllRights
   od"

definition
  create_router_cspace_object ::
  "(event option) rm_monad"
  where
  "create_router_cspace_object =
   do item ← allocate_large_item;
      create_object item RM_SelfCSpace RM_TempSlot
                    seL4_CapTableObject (of_nat CLIENT_CSPACE_SIZE_BITS)
   od"

definition
  mutate_router_cspace_cap ::
  "(event option) rm_monad"
  where
  "mutate_router_cspace_cap =
     mutate_cap RM_SelfCSpace RM_Client_CSpace
                RM_SelfCSpace RM_TempSlot
                (get_cspace_capdata (of_nat CLIENT_CSPACE_SIZE_BITS))"

definition
  create_router_ipc_object ::
  "(event option) rm_monad"
```

```
  where
  "create_router_ipc_object =
   do item ← allocate_small_item;
      create_object item RM_SelfCSpace RM_Client_IpcBuffer
                    seL4_NonArchObjectTypeCount 0
   od"

definition
  map_router_ipc_frame ::
  "(event option) rm_monad"
  where
  "map_router_ipc_frame =
   do vs ← gets (λ(tc, rm). client_vspace rm);
      map_frame vs RM_Client_IpcBuffer (of_nat CLIENT_IPC_BUFFER_VADDR)
                seL4_AllRights
   od"

definition
  create_router_tcb_object ::
  "(event option) rm_monad"
  where
  "create_router_tcb_object =
   do item ← allocate_small_item;
      create_object item RM_SelfCSpace RM_Client_Tcb
                    seL4_TCBObject 0
   od"

definition
  setup_router_tcb ::
  "(event option) rm_monad"
  where
  "setup_router_tcb =
   do vs ← gets (λ(tc, rm). client_vspace rm);
      setup_tcb RM_Client_Tcb RM_Client_CSpace
                (get_cspace_capdata (of_nat CLIENT_CSPACE_SIZE_BITS)) vs
                (of_nat CLIENT_PRIORITY) (of_nat CLIENT_IPC_BUFFER_VADDR)
                RM_Client_IpcBuffer
   od"

definition create_router_scratch_space ::
```

```
  "(event option) rm_monad"
  where
  "create_router_scratch_space =
   do item ← allocate_small_item;
      create_object item RM_SelfCSpace RM_TempSlot
                    seL4_NonArchObjectTypeCount 0
   od"

definition map_router_scratch_frame ::
  "(event option) rm_monad"
  where
  "map_router_scratch_frame =
   do vs ← gets (λ(tc, rm). client_vspace rm);
      i  ← gets (λ(tc, rm). var_i rm);
      map_frame vs RM_TempSlot
                ((of_nat CLIENT_SCRATCH_VADDR) + ((of_nat i) << seL4_PageBits))
                seL4_AllRights
   od"

definition move_router_scratch_frame ::
  "(event option) rm_monad"
  where
  "move_router_scratch_frame =
   do i ← gets (λ(tc, rm). var_i rm);
      move_cap RM_Client_CSpace
               ((of_nat CLIENT_CPTR_FIRST_SCRATCH_ITEM) + (of_nat i))
               RM_SelfCSpace RM_TempSlot
   od"

definition set_router_tcb_ip_sp ::
  "(event option) rm_monad"
  where
  "set_router_tcb_ip_sp =
   set_tcb_ip_sp RM_Client_Tcb (of_nat CLIENT_ENTRY_POINT) 0"

definition copy_timer_to_client ::
  "(event option) rm_monad"
  where
  "copy_timer_to_client =
   do ep ← gets (λ(tc, rm). virtualirq_endpoint rm);
```

```
    copy_cap RM_Client_CSpace (of_nat CLIENT_CPTR_TIMER_ENDPOINT)
             RM_SelfCSpace ep seL4_AllRights
 od"
```

definition

```
 copy_untyped_to_client ::
 "(event option) rm_monad"
 where
 "copy_untyped_to_client =
  do i ← gets (λ(tc, rm). var_i rm);
     item ← allocate_large_item;
     copy_cap RM_Client_CSpace
              ((of_nat CLIENT_CPTR_FIRST_UNTYPED_ITEM) + (of_nat i)) RM_SelfCSpace
              item seL4_AllRights
  od"
```

definition

```
 copy_asid_pool_to_client ::
 "(event option) rm_monad"
 where
 "copy_asid_pool_to_client =
  copy_cap RM_Client_CSpace (of_nat CLIENT_CPTR_ASID_POOL)
           RM_SelfCSpace RM_AsidPool seL4_AllRights"
```

definition

```
 copy_tcb_cap_to_client ::
 "(event option) rm_monad"
 where
 "copy_tcb_cap_to_client =
  move_cap RM_Client_CSpace (of_nat CLIENT_CPTR_TCB)
           RM_SelfCSpace RM_Client_Tcb"
```

definition

```
 move_ipc_buffer_cap_to_client ::
 "(event option) rm_monad"
 where
 "move_ipc_buffer_cap_to_client =
  move_cap RM_Client_CSpace (of_nat CLIENT_CPTR_IPC_BUFFER)
           RM_SelfCSpace RM_Client_IpcBuffer"
```

definition

```
mutate_cspace_to_client ::
"(event option) rm_monad"
where
"mutate_cspace_to_client =
 mutate_cap RM_SelfCSpace (of_nat CLIENT_CPTR_CSPACE)
            RM_SelfCSpace RM_Client_CSpace
            (get_cspace_capdata (of_nat CLIENT_CSPACE_SIZE_BITS))"
```

definition

```
revoke_ith_untyped ::
"(event option) rm_monad"
where
"revoke_ith_untyped =
 do i ← gets (λ(tc, rm). var_i rm);
    free_items' ← gets (λ(tc, rm). free_items rm);
    untyped_items' ← gets (λ(tc, rm). untyped_items rm);
    revoke_untyped RM_SelfCSpace
      ((untyped_items'!i)!(unat (free_items'!i)))
 od"
```

definition

```
wait_for_message ::
"(event option) rm_monad"
where
"wait_for_message =
 do command_endpoint' ← gets (λ(tc, rm). command_endpoint rm);
    lift_tc_rm (seL4_Wait command_endpoint' 0)
 od"
```

A.6 Router Manager program

definition

```
router_manager ::
"(event option) rm_monad list"
where
"router_manager =
[
(*LINE 0*)    k_instr wait_for_message,
              u_instr get_message,
```

```
(********** CASE 0 **********)
              cnd_jmp (λ(tc,rm). message rm = 0) 4,
              jmp 13,
(*** TEARDOWN ***)
(*LINE 4*)    u_instr (modify (λ(tc, rm). (tc, rm(|var_i:=0|)))),
              jmp 10,
(*LINE 6*)    cnd_jmp (λ(tc, rm). (free_items rm)!(var_i rm) ≥
                (total_items rm)!(var_i rm)) 10,
              k_instr revoke_ith_untyped,
              u_instr (modify (λ(tc, rm). (tc, free_items_update
                (λx. x[(var_i rm) := (x ! (var_i rm)) + 1]) rm))),
              jmp 6,
(*LINE 10*)   cnd_jmp (λ(tc, rm). var_i rm < NUM_SIZES) 6,
              k_instr sanitise_network_cards,
              jmp 86,

(********** CASE 1 **********)
(*LINE 13*)   cnd_jmp (λ(tc, rm). message rm = 1) 15,
              jmp 50,
(*** TEARDOWN ***)
(*LINE 15*)   u_instr (modify (λ(tc, rm). (tc, rm(|var_i:=0|)))),
              jmp 21,
(*LINE 17*)   cnd_jmp (λ(tc, rm). (free_items rm)!(var_i rm) ≥
                (total_items rm)!(var_i rm)) 21,
              k_instr revoke_ith_untyped,
              u_instr (modify (λ(tc, rm). (tc, free_items_update
                (λx. x[(var_i rm) := x!(var_i rm) + 1]) rm))),
              jmp 17,
(*LINE 21*)   cnd_jmp (λ(tc, rm). var_i rm < NUM_SIZES) 17,
              k_instr sanitise_network_cards,
(*** SETUP ***)
              k_instr create_router_cspace_object,
              k_instr mutate_router_cspace_cap,
              u_instr (modify (λ(tc, rm). (tc, rm(|var_i:=0|)))),
              jmp 31,
(*LINE 27*)   k_instr create_router_scratch_space,
              k_instr map_router_scratch_frame,
              k_instr move_router_scratch_frame,
              u_instr (modify (λ(tc, rm). (tc, rm(|var_i:=var_i rm + 1|)))),
```

```
(*LINE 31*)   cnd_jmp (λ(tc, rm). var_i rm < CLIENT_NUM_SCRATCH_FRAMES) 27,

              k_instr create_router_ipc_object,
              k_instr map_router_ipc_frame,

              k_instr create_router_tcb_object,
              k_instr setup_router_tcb,
              k_instr set_router_tcb_ip_sp,

              k_instr copy_timer_to_client,
              u_instr (modify (λ(tc, rm). (tc, rm(|var_i:=0|)))),
              jmp 42,
(*LINE 40*)   k_instr copy_untyped_to_client,
              u_instr (modify (λ(tc, rm). (tc, rm(|var_i:=var_i rm + 1|)))),
(*LINE 42*)   cnd_jmp (λ(tc, rm). var_i rm < CLIENT_NUM_UNTYPED_ITEMS) 40,

              k_instr copy_asid_pool_to_client,

              k_instr copy_nic_A_to_client,
              k_instr copy_nic_D_to_client,

              k_instr copy_tcb_cap_to_client,
              k_instr move_ipc_buffer_cap_to_client,
              k_instr mutate_cspace_to_client,
              jmp 86,

(********** CASE 2 **********)
(*LINE 50*)   cnd_jmp (λ(tc, rm). message rm = 2) 52,
              jmp 86,
(***** TEARDOWN *****)
(*LINE 52*)   u_instr (modify (λ(tc, rm). (tc, rm(|var_i:=0|)))),
              jmp 58,
(*LINE 54*)   cnd_jmp (λ(tc, rm). (free_items rm)!(var_i rm) ≥
                (total_items rm)!(var_i rm)) 58,
              k_instr revoke_ith_untyped,
              u_instr (modify (λ(tc, rm). (tc, free_items_update
                (λx. x[(var_i rm) := x!(var_i rm) + 1]) rm))),
              jmp 54,
(*LINE 58*)   cnd_jmp (λ(tc, rm). var_i rm < NUM_SIZES) 54,
              k_instr sanitise_network_cards,
```

```
(***** SETUP *****)
              k_instr create_router_cspace_object,
              k_instr mutate_router_cspace_cap,

              u_instr (modify (λ(tc, rm). (tc, rm(|var_i:=0|)))),
              jmp 68,
(*LINE 64*)   k_instr create_router_scratch_space,
              k_instr map_router_scratch_frame,
              k_instr move_router_scratch_frame,
              u_instr (modify (λ(tc, rm). (tc, rm(|var_i:=0|)))),
(*LINE 68*)   cnd_jmp (λ(tc, rm). var_i rm < CLIENT_NUM_SCRATCH_FRAMES) 64,

              k_instr create_router_ipc_object,
              k_instr map_router_ipc_frame,

              k_instr create_router_tcb_object,
              k_instr setup_router_tcb,
              k_instr set_router_tcb_ip_sp,

              k_instr copy_timer_to_client,
              u_instr (modify (λ(tc, rm). (tc, rm(|var_i:=0|)))),
              jmp 79,
(*LINE 77*)   k_instr copy_untyped_to_client,
              u_instr (modify (λ(tc, rm). (tc, rm(|var_i:=var_i rm + 1|)))),
(*LINE 79*)   cnd_jmp (λ(tc, rm). var_i rm < CLIENT_NUM_UNTYPED_ITEMS) 77,

              k_instr copy_asid_pool_to_client,

              k_instr copy_nic_A_to_client,
              k_instr copy_nic_D_to_client,

              k_instr copy_tcb_cap_to_client,
              k_instr move_ipc_buffer_cap_to_client,
              k_instr mutate_cspace_to_client,

(*LINE 86*)   jmp 0
  ]"
```

end

Appendix B

libseL4 Specification

theory *libseL4Interface*

imports *seL4Interface*

begin

B.1 libseL4 Interface

definition

```
  seL4_CNode_Copy ::
  "seL4_CPtr ⇒ seL4_CPtr ⇒ seL4_Word ⇒ seL4_CPtr ⇒ seL4_CPtr ⇒ seL4_Word
   ⇒ seL4_CapRights
   ⇒ (event option) tc_monad"
  where
  "seL4_CNode_Copy service dest_index dest_depth src_root src_index src_depth rights=
     do seL4_SetMR 0 dest_index;
        seL4_SetMR 1 (dest_depth && 0xff);
        seL4_SetMR 2 src_index;
        seL4_SetMR 3 (src_depth && 0xff);
        seL4_SetMR 4 rights;
        seL4_Call service (seL4_MessageInfo 5 1 (of_nat (unat CNodeCopy)) 0)
     od"
```

definition

```
  seL4_CNode_Move ::
  "seL4_CPtr ⇒ seL4_Word ⇒ seL4_Word ⇒ seL4_CPtr ⇒ seL4_CPtr ⇒ seL4_Word
```

```
 ⇒ (event option) tc_monad"
  where
"seL4_CNode_Move service dest_index dest_depth src_root src_index src_depth =
   do seL4_SetCap 0 src_root;
      seL4_SetMR 0 dest_index;
      seL4_SetMR 1 (dest_depth && 0xff);
      seL4_SetMR 2 src_index;
      seL4_SetMR 3 (src_depth && 0xff);
      seL4_Call service (seL4_MessageInfo 4 1 (of_nat (unat CNodeMove)) 0)
   od"
```

definition

```
  seL4_Untyped_Retype ::
"seL4_CPtr ⇒ seL4_Word ⇒ seL4_Word ⇒ seL4_CPtr ⇒ seL4_Word ⇒ seL4_Word
 ⇒ seL4_CPtr ⇒ seL4_Word
 ⇒ (event option) tc_monad"
  where
"seL4_Untyped_Retype service type size_bits root node_index node_depth node_offset
   node_window =
   do seL4_SetCap 0 root;
      seL4_SetMR 1 size_bits;
      seL4_SetMR 2 node_index;
      seL4_SetMR 3 node_depth;
      seL4_SetMR 4 node_offset;
      seL4_SetMR 5 node_window;
      seL4_Call service (seL4_MessageInfo 6 1 (of_nat (unat UntypedRetype)) 0)
   od"
```

definition

```
  seL4_CNode_Revoke ::
"seL4_CNode ⇒ seL4_Word ⇒ 8 word
 ⇒ (event option) tc_monad"
  where
"seL4_CNode_Revoke service index depth =
   do seL4_SetMR 0 index;
      seL4_SetMR 1 (of_nat (unat depth) && 0xff);
      seL4_Call service (seL4_MessageInfo 2 0 (of_nat (unat CNodeRevoke)) 0)
   od"
```

definition

```
  seL4_CNode_Mutate ::
  "seL4_CNode ⇒ seL4_Word ⇒ 8 word ⇒ seL4_CNode ⇒ seL4_Word ⇒ (8 word)
   ⇒ seL4_CapData
   ⇒ (event option) tc_monad"
  where
  "seL4_CNode_Mutate service dest_index dest_depth src_root src_index src_depth
     badge =
     do seL4_SetCap 0 src_root;
        seL4_SetMR 0 dest_index;
        seL4_SetMR 1 (of_nat (unat dest_depth) && 0xff);
        seL4_SetMR 2 src_index;
        seL4_SetMR 3 (of_nat (unat src_depth) && 0xff);
        seL4_SetMR 4 badge;
        seL4_Call service (seL4_MessageInfo 5 1 (of_nat (unat CNodeMutate)) 0)
     od"

definition
  seL4_ARM_Page_Map ::
  "seL4_ARM_Page ⇒ seL4_ARM_PageDirectory ⇒ seL4_Word ⇒ seL4_CapRights
   ⇒ seL4_ARM_VMAttributes
   ⇒ (event option) tc_monad"
  where
  "seL4_ARM_Page_Map service pd vaddr rights attr =
     do seL4_SetCap 0 pd;
        seL4_SetMR 0 vaddr;
        seL4_SetMR 1 rights;
        seL4_SetMR 2 attr;
        seL4_Call service (seL4_MessageInfo 3 1 (of_nat (unat ARMPageMap)) 0)
     od"

definition seL4_TCB_Configure ::
"seL4_TCB ⇒ seL4_Word ⇒ 8 word ⇒ seL4_CNode ⇒ seL4_CapData ⇒ seL4_CNode
 ⇒ seL4_CapData ⇒ seL4_Word ⇒ seL4_CPtr
 ⇒ (event option) tc_monad"
  where
  "seL4_TCB_Configure service fault_ep priority cspace_root cspace_root_data
     vspace_root vspace_root_data buffer bufferFrame =
     do seL4_SetCap 0 cspace_root;
        seL4_SetCap 1 vspace_root;
        seL4_SetCap 0 bufferFrame;
```

```
      seL4_SetMR 0 fault_ep;
      seL4_SetMR 1 (of_nat (unat priority) && 0xff);
      seL4_SetMR 2 cspace_root;
      seL4_SetMR 3 vspace_root;
      seL4_SetMR 4 buffer;
      seL4_Call service (seL4_MessageInfo 5 3 (of_nat (unat TCBConfigure)) 0)
    od"

definition
  seL4_TCB_WriteRegisters ::
  "seL4_TCB ⇒ bool ⇒ 8 word ⇒ seL4_Word ⇒ seL4_UserContext
   ⇒ (event option) tc_monad"
  where
  "seL4_TCB_WriteRegisters service resume_target arch_flags count regs =
     do seL4_SetMR 0
          (of_nat (unat ((of_bl::bool list ⇒ 1 word) [resume_target])) && 0x1
          || ((of_nat (unat arch_flags) && 0xff) << 8));
        seL4_SetMR 1 count;
        seL4_SetMR 2 (upc regs);
        seL4_SetMR 3 (usp regs);
        seL4_SetMR 4 (ucpsr regs);
        seL4_SetMR 5 (ur0 regs);
        seL4_SetMR 6 (ur1 regs);
        seL4_SetMR 7 (ur8 regs);
        seL4_SetMR 8 (ur9 regs);
        seL4_SetMR 9 (ur10 regs);
        seL4_SetMR 10 (ur11 regs);
        seL4_SetMR 11 (ur12 regs);
        seL4_SetMR 12 (ur4 regs);
        seL4_SetMR 13 (ur2 regs);
        seL4_SetMR 14 (ur3 regs);
        seL4_SetMR 15 (ur5 regs);
        seL4_SetMR 16 (ur6 regs);
        seL4_SetMR 17 (ur7 regs);
        seL4_SetMR 18 (ur14 regs);
        seL4_Call service (seL4_MessageInfo 19 0 (of_nat (unat TCBWriteRegisters)) 0)
      od"

end
theory seL4Interface
```

imports *libseL4Functions*

begin

B.2 seL4 Interface

```
definition seL4_Wait ::
  "seL4_CPtr ⇒ seL4_Word ⇒ (event option) tc_monad"
  where
  "seL4_Wait src sender =
     do modify (λ(i,u). (i, u (R0 := src)));
        modify (λ(i,u). (i, u (R7 := seL4_SysWait)));
        return (Some (SyscallEvent SysWait))
     od"

definition seL4_Call ::
  "seL4_CPtr ⇒ seL4_MessageInfo ⇒ (event option) tc_monad"
  where
  "seL4_Call dest msgInfo =
     do modify (λ(i,u). (i, u (R0 := dest)));
        modify (λ(i,u). (i, u (R1 := msgInfo)));
        MR0 ← gets (seL4_GetMR 0);
        modify (λ(i,u). (i, u (R2 := MR0)));
        MR1 ← gets (seL4_GetMR 1);
        modify (λ(i,u). (i, u (R3 := MR1)));
        MR2 ← gets (seL4_GetMR 2);
        modify (λ(i,u). (i, u (R4 := MR2)));
        MR3 ← gets (seL4_GetMR 3);
        modify (λ(i,u). (i, u (R5 := MR3)));
        modify (λ(i,u). (i, u (R7 := seL4_SysCall)));
        return (Some (SyscallEvent SysCall))
     od"

end
theory libseL4Functions
imports libseL4Types
begin
```

B.3 libseL4 Functions

```
definition seL4_GetMR ::
  "nat ⇒ thread_abstract_context ⇒ seL4_Word"
  where
  "seL4_GetMR n = (λ(i, u). (ipc_msg i) ! n)"

definition seL4_SetMR ::
  "nat ⇒ seL4_Word ⇒ (event option) tc_monad"
  where
  "seL4_SetMR n w =
     do modify (λ(i, u). ((ipc_msg_update (λx. x[n := w]) i, u)));
        return None
     od"

definition seL4_SetCap ::
  "nat ⇒ seL4_CPtr ⇒ (event option) tc_monad"
  where
  "seL4_SetCap n w =
     do modify (λ(i, u). ((ipc_caps_update (λx. x[n := w]) i, u)));
        return None
     od"

end
theory libseL4Types

imports Main Event_H UserADTs

begin
```

B.4 libseL4 Types

```
types seL4_Word = word32
types seL4_CPtr = seL4_Word

definition seL4_WordBits :: "nat" where "seL4_WordBits = 32"

definition seL4_PageBits :: "nat" where "seL4_PageBits = 12"
definition seL4_SlotBits :: "nat" where "seL4_SlotBits = 4"
definition seL4_TCBBits :: "nat" where "seL4_TCBBits = 9"
```

```
definition seL4_EndpointBits :: "nat" where "seL4_EndpointBits = 4"
definition seL4_PageTableBits :: "nat" where "seL4_PageTableBits = 10"
definition seL4_PageDirBits :: "nat" where "seL4_PageDirBits = 14"
definition seL4_ASIDPoolBits :: "nat" where "seL4_ASIDPoolBits = 12"

definition seL4_Frame_Args :: "nat" where "seL4_Frame_Args = 4"
definition seL4_Frame_MRs :: "nat" where "seL4_Frame_MRs = 7"
definition seL4_Frame_HasNPC :: "nat" where "seL4_Frame_HasNPC = 0"

types seL4_ARM_Page = seL4_CPtr
types seL4_ARM_PageTable = seL4_CPtr
types seL4_ARM_PageDirectory = seL4_CPtr
types seL4_ARM_ASIDControl = seL4_CPtr
types seL4_ARM_ASIDPool = seL4_CPtr


record seL4_UserContext =

  upc :: seL4_Word
  usp :: seL4_Word
  ucpsr :: seL4_Word
  ur0 :: seL4_Word
  ur1 :: seL4_Word
  ur8 :: seL4_Word
  ur9 :: seL4_Word
  ur10 :: seL4_Word
  ur11 :: seL4_Word
  ur12 :: seL4_Word
  ur4 :: seL4_Word

  ur2 :: seL4_Word
  ur3 :: seL4_Word
  ur5 :: seL4_Word
  ur6 :: seL4_Word
  ur7 :: seL4_Word
  ur14 :: seL4_Word


types seL4_ARM_VMAttributes = seL4_Word
definition seL4_ARM_PageCacheable :: "seL4_Word" where "seL4_ARM_PageCacheable = 1"
```

```
definition seL4_ARM_ParityEnabled :: "seL4_Word" where "seL4_ARM_ParityEnabled = 2"
definition seL4_ARM_Default_VMAttributes :: "seL4_Word" where
  "seL4_ARM_Default_VMAttributes = 3"



types seL4_Syscall_ID = seL4_Word
definition seL4_SysSend :: "seL4_Word" where "seL4_SysSend = -1"
definition seL4_SysNBSend :: "seL4_Word" where "seL4_SysNBSend = -2"
definition seL4_SysCall :: "seL4_Word" where "seL4_SysCall = -3"
definition seL4_SysWait :: "seL4_Word" where "seL4_SysWait = -4"
definition seL4_SysReply :: "seL4_Word" where "seL4_SysReply = -5"
definition seL4_SysReplyWait :: "seL4_Word" where "seL4_SysReplyWait = -6"
definition seL4_SysYield :: "seL4_Word" where "seL4_SysYield = -7"
definition seL4_SysDebugPutChar :: "seL4_Word" where "seL4_SysDebugPutChar = -8"
definition seL4_SysDebugHalt :: "seL4_Word" where "seL4_SysDebugHalt = -9"
definition seL4_SysDebugSnapshot :: "seL4_Word" where "seL4_SysDebugSnapshot = -10"
definition seL4_SysDebugRun :: "seL4_Word" where "seL4_SysDebugRun = -11"


types seL4_CapData = seL4_Word

definition
  seL4_CapData ::
  "5 word ⇒ 18 word ⇒ seL4_CapData"
  where
  "seL4_CapData guard_size guard_bits =
       ((of_nat (unat (0::3 word))) << 0)
     + ((of_nat (unat guard_size)) << 3)
     + ((of_nat (unat guard_bits)) << 8)"


types seL4_MessageInfo = seL4_Word

definition
  seL4_MessageInfo ::
  "7 word ⇒ 2 word ⇒ 3 word ⇒ 20 word ⇒ seL4_MessageInfo"
  where
  "seL4_MessageInfo l extraCaps capsUnwrapped label =
       ((of_nat (unat l)) << 0)
```

```
    + ((of_nat (unat extraCaps)) << 7)
    + ((of_nat (unat capsUnwrapped)) << 9)
    + ((of_nat (unat label)) << 12)"
```

```
types seL4_ObjectType = seL4_Word
definition seL4_UntypedObject :: "seL4_Word" where "seL4_UntypedObject = 0"
definition seL4_TCBObject :: "seL4_Word" where "seL4_TCBObject = 1"
definition seL4_EndpointObject :: "seL4_Word" where "seL4_EndpointObject = 2"
definition seL4_AsyncEndpointObject ::
  "seL4_Word" where "seL4_AsyncEndpointObject = 3"
definition seL4_CapTableObject :: "seL4_Word" where "seL4_CapTableObject = 4"
definition seL4_NonArchObjectTypeCount :: "seL4_Word" where
  "seL4_NonArchObjectTypeCount = 5"
```

```
types seL4_CapRights = seL4_Word
definition seL4_CanWrite :: "seL4_Word" where "seL4_CanWrite = 1"
definition seL4_CanRead :: "seL4_Word" where "seL4_CanRead = 2"
definition seL4_CanGrant :: "seL4_Word" where "seL4_CanGrant = 4"
definition seL4_AllRights :: "seL4_Word" where "seL4_AllRights = 7"
definition seL4_Transfer_Mint :: "seL4_Word" where "seL4_Transfer_Mint = 256"
```

```
datatype seL4_IPCBuffer =
    seL4_IPCBuffer seL4_MessageInfo "(seL4_Word list)" seL4_Word
      "(seL4_CPtr list)" seL4_CPtr seL4_CPtr seL4_Word

definition
  ipc_msg ::
  "seL4_IPCBuffer ⇒ seL4_Word list"
  where
  "ipc_msg =
     (λx. (case x of
             seL4_IPCBuffer tag msg userData caps receiveCNode
               receiveIndex receiveDepth
               ⇒ msg))"
```

definition

```
  ipc_msg_update ::
  "(seL4_Word list ⇒ seL4_Word list) ⇒ seL4_IPCBuffer ⇒ seL4_IPCBuffer"
  where
  "ipc_msg_update f =
     (λx. (case x of
               seL4_IPCBuffer tag msg userData caps receiveCNode
                 receiveIndex receiveDepth
                 ⇒ seL4_IPCBuffer tag (f msg) userData caps receiveCNode
                     receiveIndex receiveDepth))"
```

definition

```
  ipc_caps ::
  "seL4_IPCBuffer ⇒ seL4_Word list"
  where
  "ipc_caps =
     (λx. (case x of
             seL4_IPCBuffer tag msg userData caps receiveCNode
               receiveIndex receiveDepth
               ⇒ caps))"
```

definition

```
  ipc_caps_update ::
  "(seL4_Word list ⇒ seL4_Word list)
  ⇒ seL4_IPCBuffer
  ⇒ seL4_IPCBuffer"
  where
  "ipc_caps_update f =
     (λx. (case x of
               seL4_IPCBuffer tag msg userData caps receiveCNode
                 receiveIndex receiveDepth
                 ⇒ seL4_IPCBuffer tag msg userData (f caps) receiveCNode
                     receiveIndex receiveDepth))"
```

types *seL4_CNode = seL4_CPtr*
types *seL4_IRQHandler = seL4_CPtr*
types *seL4_IRQControl = seL4_CPtr*
types *seL4_TCB = seL4_CPtr*
types *seL4_Untyped = seL4_CPtr*

definition *InvalidInvocation :: "seL4_Word"* **where** *"InvalidInvocation = 0"*
definition *UntypedRetype :: "seL4_Word"* **where** *"UntypedRetype = 1"*
definition *TCBReadRegisters :: "seL4_Word"* **where** *"TCBReadRegisters = 2"*
definition *TCBWriteRegisters :: "seL4_Word"* **where** *"TCBWriteRegisters = 3"*
definition *TCBCopyRegisters :: "seL4_Word"* **where** *"TCBCopyRegisters = 4"*
definition *TCBConfigure :: "seL4_Word"* **where** *"TCBConfigure = 5"*
definition *TCBSetPriority :: "seL4_Word"* **where** *"TCBSetPriority = 6"*
definition *TCBSetIPCBuffer :: "seL4_Word"* **where** *"TCBSetIPCBuffer = 7"*
definition *TCBSetSpace :: "seL4_Word"* **where** *"TCBSetSpace = 8"*
definition *TCBSuspend :: "seL4_Word"* **where** *"TCBSuspend = 9"*
definition *TCBResume :: "seL4_Word"* **where** *"TCBResume = 10"*
definition *CNodeRevoke :: "seL4_Word"* **where** *"CNodeRevoke = 11"*
definition *CNodeDelete :: "seL4_Word"* **where** *"CNodeDelete = 12"*
definition *CNodeRecycle :: "seL4_Word"* **where** *"CNodeRecycle = 13"*
definition *CNodeCopy :: "seL4_Word"* **where** *"CNodeCopy = 14"*
definition *CNodeMint :: "seL4_Word"* **where** *"CNodeMint = 15"*
definition *CNodeMove :: "seL4_Word"* **where** *"CNodeMove = 16"*
definition *CNodeMutate :: "seL4_Word"* **where** *"CNodeMutate = 17"*
definition *CNodeRotate :: "seL4_Word"* **where** *"CNodeRotate = 18"*
definition *CNodeSaveCaller :: "seL4_Word"* **where** *"CNodeSaveCaller = 19"*
definition *IRQIssueIRQHandler :: "seL4_Word"* **where** *"IRQIssueIRQHandler = 20"*
definition *IRQInterruptControl :: "seL4_Word"* **where** *"IRQInterruptControl = 21"*
definition *IRQAckIRQ :: "seL4_Word"* **where** *"IRQAckIRQ = 22"*
definition *IRQSetIRQHandler :: "seL4_Word"* **where** *"IRQSetIRQHandler = 23"*
definition *IRQClearIRQHandler :: "seL4_Word"* **where** *"IRQClearIRQHandler = 24"*
definition *nInvocationLabels :: "seL4_Word"* **where** *"nInvocationLabels = 25"*

definition *ARMPageTableMap :: "seL4_Word"* **where** *"ARMPageTableMap = nInvocationLabels"*
definition *ARMPageTableUnmap :: "seL4_Word"* **where**
"ARMPageTableUnmap = nInvocationLabels + 1"
definition *ARMPageMap :: "seL4_Word"* **where** *"ARMPageMap = nInvocationLabels + 2"*
definition *ARMPageRemap :: "seL4_Word"* **where** *"ARMPageRemap = nInvocationLabels + 3"*
definition *ARMPageUnmap :: "seL4_Word"* **where** *"ARMPageUnmap = nInvocationLabels + 4"*
definition *ARMPageFlushCaches :: "seL4_Word"* **where**
"ARMPageFlushCaches = nInvocationLabels + 5"

definition *ARMASIDControlMakePool :: "seL4_Word"* **where**
"ARMASIDControlMakePool = nInvocationLabels + 6"
definition *ARMASIDPoolAssign :: "seL4_Word"* **where**
"ARMASIDPoolAssign = nInvocationLabels + 7"

types *thread_abstract_context = "seL4_IPCBuffer × user_context"*
types *'a tc_monad = "(thread_abstract_context, 'a) nondet_monad"*

end

Appendix C

System Process

theory *Decode_I*

imports *RouterManager_I*

begin

C.1 Router Manager Functional specification system process

C.1.1 Lifting from rm_monad to user_state transitions

consts

```
  decode_sac_state_I ::
  "user_mem ⇒ sac_abstract_state"
```

consts

```
  decode_ipc_I ::
  "user_mem ⇒ seL4_IPCBuffer"
```

definition

```
  decode_I ::
  "user_state ⇒ thread_abstract_context × sac_abstract_state"
  where
  "decode_I = (λ(uc, ip, um).
    ((decode_ipc_I um, uc), ip_update (λx. ip >> 2) (decode_sac_state_I um))) "
```

consts

```
encode_sac_state_I ::
"seL4_IPCBuffer ⇒ sac_abstract_state ⇒ user_mem"
```

definition

```
encode_I ::
"thread_abstract_context × sac_abstract_state ⇒ user_state"
```

where

```
"encode_I = (λ((ipc, uc), s). (uc, (ip s << 2), encode_sac_state_I ipc s))"
```

definition

```
encode_return_I ::
"((event option) × thread_abstract_context × sac_abstract_state) set × bool
 ⇒ (user_state × event option) set"
```

where

```
"encode_return_I = (λ(Q,b). {(encode_I s, e)|s e. (e, s) ∈ Q})"
```

definition

```
lift_I ::
"instr ⇒ (user_state ⇒ (user_state × event option) set)"
```

where

```
"lift_I i = (λs. encode_return_I (i (decode_I s)))"
```

C.2 User state translations

consts

```
program_text_I ::
"word32 ⇒ word8 option"
```

consts

```
project_program_text ::
"user_state ⇒ (word32 ⇀ word8)"
```

consts

```
decode_program_text ::
"(word32 ⇀ word8)
 ⇒ (user_state ⇒ (user_state × event option) set) list"
```

consts

```
project_program_counter ::
"user_state ⇒ nat"
```

axioms *decode_router_manager_program_text:*
"decode_program_text program_text_I = map lift_I router_manager"

C.3 The user specification

definition
uop_I ::
"user_state ⇒ (user_state × event option) set"
where
"uop_I us =
((decode_program_text (project_program_text us))!(project_program_counter us)) us"

C.4 The system process

definition
ADT_I :: "(state global_state, observable, global_transition) data_type"
where
"ADT_I ≡
(| Init = λs. Init_A, Fin = Fin_A,
Step = global_automaton (do_user_op uop_I) kernel_call_A |)"

end

theory *UserADTs*

imports *Simulation Invariant*

begin

C.5 System process

We constrain idle thread behaviour, so we distinguish three main machine control states:

datatype *mode = UserMode | KernelMode | IdleMode*

types *user_mem = "word32 ⇒ word8 option"*
types *user_state = "user_context × word32 × user_mem"*

types *observable_user_state = "user_context × user_mem"*
types *machine_no_mem = "irq set × machine_state_rest"*

We take as observable state the user state (register context and memory), the internal machine state without memory, the current-thread pointer in both specs (i.e. *cur_thread* in the abstract kernel model and *ksCurThread* in the executable specification), the mode, and the current event (if any).

Currently, user memory is all memory accessible to any user (everything of type *UserData*). A virtual-memory view could be built on top of this.

types *observable = "observable_user_state × machine_no_mem × word32 × mode × event option"*

The global state contains the current register context of the machine as well as the internal kernel state, the mode and the current event (if any).

types *'k global_state = "(user_context × word32 × 'k) × mode × event option"*

The global-transition type reflects the operations of the abstract data types.

datatype
global_transition =
KernelTransition
| UserTransition
| UserEventTransition
| IdleEventTransition

C.6 The global_automaton

The following definition models machine, user, and kernel entry/exit behaviour abstractly. It also constrains the behaviour of the idle thread. The first parameter specifies all possible user transition on the kernel state (which is expected to inject the new user memory into the kernel state) and the second parameter provides a kernel specification (it will later be used with the abstract as well as the executable specification).

The first and second transition sets are kernel calls being serviced by the kernel. In the first set the kernel exits into UserMode, in the second the kernel exits into IdleMode.

The third transition set is normal user execution.

The fourth transition set is user execution that generates a syscall event and traps into the kernel.

The fifth transition set is user execution being interrupted by an interrupt event.

The sixth transition set is idle execution.

definition

```
global_automaton ::
  "((user_context × word32 × 'k) × (user_context × word32 × 'k)
    × mode × event option) set
   ⇒ (event ⇒ ((user_context × 'k) × mode × (user_context × 'k)) set)
   ⇒ global_transition ⇒ ('k global_state × 'k global_state) set"
```

where

```
"global_automaton user_transition kernel_call t ≡ case t of
  KernelTransition ⇒
     { ( ((uc, ip, s), KernelMode, Some e),
         ((uc', uc' LR_svc, s'), UserMode, None) ) |uc ip s uc' s' e.
           ((uc, s), UserMode, (uc', s')) ∈ kernel_call e }
   ∪ { ( ((uc, ip, s), KernelMode, Some e),
         ((uc', ip', s'), IdleMode, None) ) |uc ip s uc' ip' s' e.
           ((uc, s), IdleMode, (uc', s')) ∈ kernel_call e }
| UserTransition ⇒
     { ( (g, UserMode, None), (g', UserMode, e) ) |g g' e.
     (g, g', UserMode, e) ∈ user_transition}
   ∪ { ( ((uc, ip, g), UserMode, None),
         ((uc'(FaultInstruction := ip'), ip', g'), KernelMode, Some ev) )
         |uc ip g uc' ip' g' ev.
           ((uc, ip, g), (uc', ip', g'), KernelMode, Some ev) ∈ user_transition}
   ∪ { ( (g, UserMode, None), (g, KernelMode, Some Interrupt) ) |g. True }
| IdleEventTransition ⇒
   { ( (g, IdleMode, None), (g, KernelMode, Some Interrupt) ) |g. True }"
```

After kernel initialisation, the machine is in UserMode, running the initial thread.

definition

```
  Init_A :: "state global_state set"
```

where

```
  "Init_A ≡ {((empty_context, 0, init_A_st), UserMode, None)}"
```

A pointer is inside a user frame if its top bits point to any object of type `IntData`

definition

```
  "in_user_frame p s ≡ ∃sz. typ_at (AArch (AIntData sz)) (p && ~~ mask (pageBitsForSize sz))
s"
```

The content of user memory is stored in the machine state. Only locations tagged as inside a user frame are accessible.

definition

```
  "user_mem s ≡ λp.
```

```
if in_user_frame p s
then Some (underlying_memory (machine_state s) p)
else None"
```

definition

```
"user_memory_update um ≡ modify (λms.
 ms(|underlying_memory := (λa. case um a of Some x ⇒ x
                              | None ⇒ underlying_memory ms a)|))"
```

definition

```
do_user_op' ::
"  (user_state ⇒ (user_state × event option) set)
 ⇒ user_context × word32
 ⇒ ((user_context × word32) × mode × event option) s_monad"
where
"do_user_op' uop ≡ (λ(uc, ip).
   do um ← gets user_mem;
      ((uc',ip',um'), e) ← select (uop (uc,ip,um));
      m ← (case e of None ⇒ return UserMode | Some ev ⇒ return KernelMode);
      do_machine_op (user_memory_update um');
      return ((uc',ip'), m, e)
   od)"
```

definition

```
do_user_op ::
" (user_state ⇒ (user_state × event option) set)
⇒ ((user_context × word32 × state) × ((user_context × word32 × state) × mode
    × event option)) set"
where
"do_user_op uop ≡
   {((uc, ip, s), ((uc', ip', s'), m, e)).
     (((uc', ip'), m, e), s') ∈ fst (split (do_user_op' uop) ((uc, ip), s)) }"
```

Kernel calls are described completely by the abstract and concrete spec. We model kernel entry by allowing an arbitrary user (register) context. The mode after a kernel call is either user or idle (see also thm in Refine.thy).

definition

```
kernel_entry :: "event ⇒ user_context ⇒ user_context s_monad"
where
"kernel_entry e tc ≡ do
```

```
  t ← gets cur_thread;
  thread_set (λtcb. tcb (| tcb_context := tc |)) t;
  call_kernel e;
  t' ← gets cur_thread;
  thread_get tcb_context t'
od"
```

definition

```
kernel_call_A ::
"event ⇒ ((user_context × state) × mode × (user_context × state)) set"
where
"kernel_call_A e ≡
    {(s, m, s'). s' ∈ fst (split (kernel_entry e) s) ∧
                 m = (if ct_running (snd s') then UserMode else IdleMode)}"
```

Extracting the observable state:

definition

```
"no_mem_machine m ≡ (irq_masks m, machine_state_rest m)"
```

The final observable state:

definition

```
Fin_A :: "state global_state ⇒ observable"
```

where

```
"Fin_A ≡ λ((uc, ip, s), m, e).
  ((uc, user_mem s), no_mem_machine (machine_state s), cur_thread s, m, e)"
```

Lifting a state relation on kernel states to global states.

definition

```
"lift_state_relation sr ≡
 { (((tc,s),m,e), ((tc,s'),m,e))|s s' m e tc. (s,s') ∈ sr }"
```

lemma *lift_state_relationD:*

```
"(((tc, s), m, e), ((tc', s'), m', e')) ∈ lift_state_relation R ⟹
(s,s') ∈ R ∧ tc' = tc ∧ m' = m ∧ e' = e"
by (simp add: lift_state_relation_def)
```

lemma *lift_state_relationI:*

```
"(s,s') ∈ R ⟹ (((tc, s), m, e), ((tc, s'), m, e)) ∈ lift_state_relation R"
by (fastsimp simp: lift_state_relation_def)
```

lemma *in_lift_state_relation_eq:*
"(((tc, s), m, e), (tc', s'), m', e') ∈ lift_state_relation R ⟷
(s, s') ∈ R ∧ tc' = tc ∧ m' = m ∧ e' = e"
by *(auto simp add: lift_state_relation_def)*

end

Bibliography

[1] J. Andronick, D. Greenaway, and K. Elphinstone. Towards proving security in the presence of large untrusted components. In G. Klein, R. Huuck, and B. Schlich, editors, *Proceedings of the 5th Workshop on Systems Software Verification*, Vancouver, Canada, Oct 2010. USENIX.

[2] D. Elkaduwe, G. Klein, and K. Elphinstone. Verified protection model of the seL4 microkernel. In J. Woodcock and N. Shankar, editors, *Proceedings of Verified Software: Theories, Tools and Experiments 2008*, volume 5295 of *Lecture Notes in Computer Science*, pages 99–114, Toronto, Canada, Oct 2008. Springer-Verlag.

[3] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. *Communications of the ACM*, 53(6):107–115, Jun 2010.

[4] G. Klein, T. Murray, P. Gammie, T. Sewell, and S. Winwood. Provable security: How feasible is it? In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems*, Napa, CA, USA, May 2011.

[5] G. Klein, T. Sewell, and S. Winwood. *Refinement in the formal verification of seL4*, pages 323–339. Springer, 1st edition, March 2010.

[6] L. C. Paulson. The isabelle reference manual, 2008.

[7] S. Winwood, G. Klein, T. Sewell, J. Andronick, D. Cock, and M. Norrish. Mind the gap: A verification framework for low-level C. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *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, Aug 2009. Springer-Verlag.

Noninterference for Operating System Kernels

Toby Murray[1,2], Daniel Matichuk[1], Matthew Brassil[1], Peter Gammie[1] and Gerwin Klein[1,2]

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

{firstname.lastname}@nicta.com.au

Abstract. While intransitive noninterference is a natural property for any secure OS kernel to enforce, proving that the implementation of any particular general-purpose kernel enforces this property is yet to be achieved. In this paper we take a significant step towards this vision by presenting a machine-checked formulation of intransitive noninterference for OS kernels, and its associated sound and complete unwinding conditions, as well as a scalable proof calculus over nondeterministic state monads for discharging these unwinding conditions across a kernel's implementation. Our ongoing experience applying this noninterference framework and proof calculus to the seL4 microkernel validates their utility and real-world applicability.

Keywords: Information flow, refinement, scheduling, state monads.

1 Introduction

A primary function of any operating system (OS) kernel is to enforce security properties and policies. The classical security property of *noninterference* [8] formalises the absence of unwanted information flows within a system, and is a natural goal for any secure OS to aim to enforce. Here, the system is divided into a number of *domains*, and the allowed information flows between domains specified by means of an information flow *policy* $\rightsquigarrow$, such that $d \rightsquigarrow d'$ if information is allowed to flow from domain d to domain d'. So-called *intransitive noninterference* [10] generalises noninterference to the case in which the relation $\rightsquigarrow$ is possibly intransitive.

While intransitive noninterference is a natural property for any secure OS kernel to enforce, proving that the implementation of any particular general-purpose kernel enforces this property is yet to be achieved. In this paper we

* 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 FA2386-10-1-4105. 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.

take a significant step towards this vision by presenting a machine-checked formulation of intransitive noninterference for OS kernels, and its associated sound and complete proof obligations (called *unwinding conditions*), as well as a scalable proof calculus over nondeterministic state monads for discharging these unwinding conditions across a kernel's implementation. Both our noninterference formulation and proof calculus are termination-insensitive, under the assumption that a noninterference verification for an OS kernel is performed only after proving that its execution is always defined (and thus every system call always terminates). Our experience applying this noninterference framework and proof calculus to the seL4 microkernel [11] validates their utility and real-world applicability.

Our intransitive noninterference formulation improves on traditional formulations [10, 16, 19, 21] in two ways that make it more suitable for application to OS kernels. Firstly, traditional formulations of intransitive noninterference assume a static mapping dom from actions to domains, such that the domain dom a on whose behalf some action a is being performed can be determined solely from the action itself. No such mapping exists in the case of an OS kernel, which must infer this information at run-time. For instance, when a system call occurs, in order to work out which thread has requested the system call the kernel must consult the data-structures of the scheduler to determine which thread is currently running. This prevents traditional noninterference formulations from being able to reason about potential information flows that might occur via these scheduling data structures. An example would be a scheduler that does not properly isolate domains by basing its decision about whether to schedule a Low thread on whether a High thread is runnable. Our noninterference formulation makes dom dependent on the current state s, in order to overcome this problem, such that the domain associated with some action a that occurs from state s is dom a s. This makes the resulting noninterference formulation entirely state-dependent and complicates the proofs of soundness for our unwinding conditions. Proving that a system satisfies these unwinding conditions (and therefore our formulation of noninterference) requires showing that the scheduler does not leak information via its scheduling decisions.

Secondly, while phrased for (possibly) nondeterministic systems, our noninterference formulation is preserved by refinement. As explained later, this requires it to preclude all *domain-visible* nondeterminism, which necessarily abstracts away possible sources of information. Being preserved by refinement is important in allowing our noninterference formulation to be proved of real kernels at reasonable cost, as it can be proved about an abstract specification and then transported to the more complex implementation by refinement. In the case of seL4, this allows us to prove noninterference about a mostly-deterministic refinement [13] of its abstract *functional* specification, which its C implementation has been proved to refine [11], in order to conclude it of the implementation. Our experience to date suggests that reasoning about seL4's functional specification requires an order of magnitude less effort than reasoning directly about the implementation [12].

Our proof calculus resembles prior language-based frameworks for proving termination-insensitive confidentiality (and other relational) properties of pro-

grams [1, 2, 4]; however, it is better suited than these frameworks for general-purpose OS kernels. Firstly, our calculus aims not at generality but rather at scalability, which is essential to enable its practical application to entire OS kernels. Secondly, it is explicitly designed for reasoning about systems for which no complete static assignment of memory locations or program variables to security domains exists. As is the case with a general-purpose OS kernel like seL4 that implements a dynamic access control system, whether a memory location is allowed to be read by the currently running thread depends on the access rights that the current thread has, if any, to that location. In a microkernel like seL4 that implements virtual memory, this of course depends on the current virtual memory mappings for the currently running thread. Thus, like the mapping of actions to domains, the mapping of memory locations to domains is also state-dependent in a general-purpose OS kernel. Our proof calculus is tuned to tracking and discharging these kinds of state-dependent proof obligations that arise when reasoning about confidentiality in such a system. These manifest themselves as preconditions on confidentiality statements about individual function calls that we discharge using a monadic Hoare logic and its associated VCG [6]. Our calculus is specially tuned so that this same VCG engine can automate its application, without modification, to automatically prove confidentiality statements for those functions that do not read confidential state (i.e. the vast majority of them), given appropriate user-supplied loop invariants.

Our experience applying this calculus to seL4 suggests that it scales very well to real-world systems. So far we have used it to prove confidentiality for 98% of the functions in the abstract seL4 specification in under 15 person-months. The remaining fraction comprises nondeterministic functions that abstract away from sources of confidential information — i.e. parts of the specification that are too abstract to allow correct reasoning about confidentiality. We are in the process of making these parts of the specification more concrete to produce a refinement of the functional specification, which seL4's C implementation refines in turn, suitable for reasoning about confidentiality [13]. We have already done this and proved confidentiality for the *revoke* system call, which is the kernel's most complex code path. The remaining functions are currently in progress.

In this paper, Section 2 presents our noninterference formulation for OS kernels and its associated unwinding conditions. In Section 3 we present our proof calculus for discharging these unwinding conditions across an entire kernel. Section 4 considers related work before Section 5 concludes. All theorems and definitions in this paper have been generated directly from the interactive theorem prover Isabelle/HOL [14] in which all of our work was carried out.

2 Noninterference

Our noninterference formulation for OS kernels extends von Oheimb's notion of *noninfluence* [21]. We formalise noninterference over de Roever-Engelhardt style *data types* [7], which can be thought of as automata with a supported theory of refinement, allowing us to prove that our noninterference formulation is preserved

under refinement. We first introduce the data type formalism and the notion of refinement, before presenting our noninterference formulation and its associated unwinding conditions. We prove that the unwinding conditions are sound and complete for our noninterference formulation. We explain how our unwinding conditions (and, hence, our noninterference formulation) require us to prove the absence of information leaks through scheduling decisions. Lastly we show how our noninterference formulation is preserved by refinement.

2.1 Data Types and Refinement

We model an OS kernel as a state machine whose transitions include processing an interrupt or exception, performing a system call, and ordinary user-level actions like reading and writing user-accessible memory. We use the terms *event* and *action* interchangeably to refer to an automaton's individual transitions.

A data type automaton A is simply a triple comprising three functions: an initialisation function $\mathsf{Init}_A :: \mathit{state} \Rightarrow \mathit{istate\ set}$ that maps individual *observable* states to sets of corresponding internal states, an internal step relation $\mathsf{IStep}_A :: \mathit{event} \Rightarrow (\mathit{istate} \times \mathit{istate})\ \mathit{set}$, and a final projection function $\mathsf{Fin}_A :: \mathit{istate} \Rightarrow \mathit{state}$ that maps individual internal states to corresponding individual observable states. For a data type A and initial observable state $s :: \mathit{state}$ and sequence of transitions as, let $\mathsf{execution}\ A\ s\ as$ denote the set of observable states that can be reached by A performing as. $\mathsf{execution}\ A\ s\ as$ operates by first applying Init_A to the observable state s to produce a set of corresponding initial internal states. It then computes a set of resulting internal states by repeatedly applying IStep_A to each event a in as in turn to arrive at a set of final internal states. To each of these it applies Fin_A to obtain the set of final observable states.

$$\mathsf{execution}\ A\ s\ as \equiv \mathsf{Fin}_A\ `\ \mathsf{foldl}\ (\lambda S\ a.\ \mathsf{IStep}_A\ a\ ``\ S)\ (\mathsf{Init}_A\ s)\ as$$

Here $R\ ``\ S$ and $f\ `\ S$ are the relational images of the set S under the relation R and function f respectively, and foldl is the standard fold function on lists.

A data type C refines data type A, written $A \sqsubseteq C$, when its behaviours are a subset of A's.

$$A \sqsubseteq C \equiv \forall\, s\ as.\ \mathsf{execution}\ C\ s\ as \subseteq \mathsf{execution}\ A\ s\ as$$

2.2 System Model

Let A be an automaton, whose observable state is of type *state*, and $\mathsf{s}_0 :: \mathit{state}$ denote the initial observable state from which execution of A begins. Let $\mathsf{reachable}\ s$ denote that observable state s is reachable from s_0:

$$\mathsf{reachable}\ s \equiv \exists\, as.\ s \in \mathsf{execution}\ A\ \mathsf{s}_0\ as$$

As occurs in OS kernels generally, we assume that every event is always enabled.

$$\mathsf{reachable}\ s \longrightarrow (\exists\, s'.\ s' \in \mathsf{execution}\ A\ s\ as)$$

Let the function Step characterise the single-step behaviour of the system:

$$\mathsf{Step}\ a \equiv \{(s,\ s') \mid s' \in \mathsf{execution}\ A\ s\ [a]\}$$

For the information flow policy, we assume a set of security domains and a reflexive relation $\rightsquigarrow$ that specifies the allowed information flows between domains, where $d \rightsquigarrow d'$ implies that information is allowed to flow from domain d directly to d'. Noninterference asserts that no information flows outside of $\rightsquigarrow$ can occur.

For each domain d, let $\overset{d}{\sim}$ be an equivalence relation on observable states, such that $s \overset{d}{\sim} t$ if and only if domain d's state is identical in s and t. Here, d's state will include the user-visible state that d can directly read, but might also include kernel-level state that the kernel might legitimately reveal to d. This relation is sometimes called an *unwinding relation*. When the system transitions directly from state s to state s' and $s \overset{d}{\sim} s'$ for example, domain d has not been observably affected by this transition. For a set of domains D, let $s \overset{D}{\approx} t \equiv \forall\, d{\in}D.\ s \overset{d}{\sim} t$.

Traditional noninterference formulations associate a security domain $\mathsf{dom}\ e$ with each event e that occurs, which defines the domain that performed the event. Recall from Section 1 that when a system call event occurs, the kernel must consult the data structures of the scheduler to determine which thread performed the system call, which will be the thread that is currently active. So events are not intrinsically associated with domains; rather, this association depends on part of the current state of the system which records the currently running domain.

Therefore, let $\mathsf{dom} :: \mathit{event} \Rightarrow \mathit{state} \Rightarrow \mathit{domain}$ be a function such that $\mathsf{dom}\ e\ s$ gives the security domain that is associated with event e in state s. When the scheduler's state is identical in states s and t, we expect that $\mathsf{dom}\ e\ s = \mathsf{dom}\ e\ t$ for all events e. Formally, let $\mathsf{s\text{-}dom} :: \mathit{domain}$ be an arbitrary domain, whose state encompasses that part of the system state that determines which domain is currently active. s-dom stands for *scheduler domain*. Then we assume that for all events e and states s and t

$$s \overset{\mathsf{s\text{-}dom}}{\sim} t \longrightarrow \mathsf{dom}\ e\ s = \mathsf{dom}\ e\ t$$

Actions of the scheduling domain s-dom naturally include all those that schedule a new domain d to execute. We expect that when a domain d is scheduled, that d will be able to detect that it is now active, and so that an information flow might have occurred from s-dom to d. Since the scheduler can possibly schedule any domain, we expect that a wellformed information flow policy $\rightsquigarrow$ will have an edge from s-dom to every domain d:

$$\mathsf{s\text{-}dom} \rightsquigarrow d$$

In order to prevent s-dom from being a global transitive channel by which information can flow from any domain to any other, we require that information can never flow directly from any other domain d to s-dom, so

$$d \rightsquigarrow \mathsf{s\text{-}dom} \longrightarrow d = \mathsf{s\text{-}dom}.$$

This restriction forces us to prove that the scheduler's decisions about which domain should execute next are independent of the other domains, which is typical scheduler behaviour in a separation kernel.

2.3 Formulating Noninterference

Traditionally [16], intransitive noninterference definitions make use of a sources function, whereby for a sequence of actions as and a domain d, sources as d gives the set of domains that are allowed to pass information to d when as occurs. Because our dom function depends on the current state s, sources must do so as well. Therefore let sources as s d denote the set of domains that can pass information to d when as occurs, beginning in state s. The following definition is an extension of the standard one [16,21] in line with our augmented dom function.

$$\begin{aligned}&\mathsf{sources}\ []\ s\ d &&= \{d\}\\ &\mathsf{sources}\ (a{\cdot}as)\ s\ d = \\ &\bigcup\{\mathsf{sources}\ as\ s'\ d \mid (s,\ s') \in \mathsf{Step}\ a\} \cup \\ &\{w \mid w = \mathsf{dom}\ a\ s \wedge (\exists\, v\ s'.\ \mathsf{dom}\ a\ s \rightsquigarrow v \wedge (s,\ s') \in \mathsf{Step}\ a \wedge v \in \mathsf{sources}\ as\ s'\ d)\}\end{aligned}$$

Here, we include in sources $(a{\cdot}as)$ s d all domains that can pass information to d when as occurs from all successor-states s' of s, as well as the domain dom a s performing a, whenever there exists some intermediate domain v that it is allowed to pass information to who in turn can pass information to d when the remaining events as occur from some successor state s' of s. An alternative, and seemingly more restrictive, definition would include only those domains that are present in all sources as s' d, and include dom a s only when some such v can be found for each sources as s' d, where $(s,\ s') \in \mathsf{Step}\ a$. However, as a consequence of Lemma 2 introduced later, this yields an equivalent noninterference formulation.

As is usual, the sources function is used to define a purge function, ipurge, in terms of which noninterference is formulated. Traditionally, for a domain d and action sequence as, ipurge d as returns the sequence of actions as with all actions removed that are not allowed to (indirectly) influence d when as occurs [16]. Naturally, we must include the current state s in our ipurge function. However, for nondeterministic systems purging may proceed from a set ss of possible initial states. This leads to the following definition.

$$\begin{aligned}&\mathsf{ipurge}\ d\ []\ ss &&= []\\ &\mathsf{ipurge}\ d\ (a{\cdot}as)\ ss = &&\ \mathsf{if}\ \exists\, s{\in}ss.\ \mathsf{dom}\ a\ s \in \mathsf{sources}\ (a{\cdot}as)\ s\ d\\ &&&\ \mathsf{then}\ a{\cdot}\mathsf{ipurge}\ d\ as\ (\textstyle\bigcup_{s\in ss}\{s' \mid (s,\ s') \in \mathsf{Step}\ a\})\\ &&&\ \mathsf{else}\ \mathsf{ipurge}\ d\ as\ ss\end{aligned}$$

Initially, the set ss will be a singleton containing one initial state s. Given a sequence of actions $a{\cdot}as$ being performed from s, ipurge will keep the first action a if dom a s $\in$ sources $(a{\cdot}as)$ s d, i.e. if this action is allowed to affect the target domain d. Purging then continues on the remaining actions as from the successor states of s after a. On the other hand, if the action a being performed is not allowed to affect the target domain d, then it is removed from the sequence. For this reason, purging continues on the remaining actions as from the current state s, rather than its successors. We require the action to be able to affect the target domain in only one of the states $s \in ss$ to avoid purging it. An alternative definition would instead place this requirement on all states $s \in ss$. Again however, because of Lemma 2, this yields an equivalent noninterference formulation.

For states s and t and sequences of actions as and bs and domain d, let $\mathsf{uwr\text{-}equiv}\ s\ as\ t\ bs\ d$ denote when the contents of domain d is identical after executing as from s and bs from t in all resulting pairs of states. When $\mathsf{uwr\text{-}equiv}\ s\ as\ t\ bs\ d$ is true, domain d is unable to distinguish the cases in which as is executed from s, and bs is executed from t. Recall that we assume that every event is always enabled and that divergence never occurs on any individual execution step, under the assumption that noninterference is proved only after proving that a system's execution is always defined. This is why uwr-equiv and the following noninterference formulation are termination-insensitive.

$$\mathsf{uwr\text{-}equiv}\ s\ as\ t\ bs\ d \equiv$$
$$\forall\, s'\ t'.\ s' \in \mathsf{execution}\ A\ s\ as \wedge t' \in \mathsf{execution}\ A\ t\ bs \longrightarrow s' \overset{d}{\sim} t'$$

Traditionally [16, 21] this property is defined using a projection function $\mathsf{out} :: \mathit{domain} \Rightarrow \mathit{state} \Rightarrow \mathit{output}$ so that, rather than testing whether $s' \overset{d}{\sim} t'$ for final states s' and t', it tests whether $\mathsf{out}\ d\ s' = \mathsf{out}\ d\ t'$. However, these traditional formulations invariably require the unwinding condition of *output consistency* which asserts that $\mathsf{out}\ d\ s' = \mathsf{out}\ d\ t'$ whenever $s' \overset{d}{\sim} t'$, and construct the remaining unwinding conditions to establish precisely this latter relation. We avoid this indirection by discarding out entirely. One could re-phrase the noninterference formulation here in terms of out if necessary, in which case the addition of output consistency to the unwinding conditions presented here would be sufficient to prove the resulting noninterference property.

We now have the ingredients to express our noninterference formulation, which we derive as follows. Given two action sequences as and bs, a domain d, and an initial state s from which each sequence is executed, if $\mathsf{ipurge}\ d\ as\ \{s\} = \mathsf{ipurge}\ d\ bs\ \{s\}$ then, when all events that are not allowed to affect d are removed from each sequence, they are both identical. So if none of these removed events can actually affect d, we should expect that d cannot distinguish the execution of one sequence from the other, i.e. that $\mathsf{uwr\text{-}equiv}\ s\ as\ s\ bs\ d$ should hold.

However, the only domains that should be able to affect d when as executes here are those in $\mathsf{sources}\ as\ s\ d$. So if s were modified to produce a state t from which bs was executed instead, we should expect $\mathsf{uwr\text{-}equiv}\ s\ as\ t\ bs\ d$ to hold so long as: (1) s and t agree on the state of all domains in $\mathsf{sources}\ as\ s\ d$, i.e. $s \overset{\mathsf{sources}\ as\ s\ d}{\approx} t$, and (2) the same domain is currently active in both, i.e. $s \overset{\mathsf{s\text{-}dom}}{\sim} t$. This is our formulation of von Oheimb's *noninfluence* [21], denoted noninfluence.

$$\begin{aligned}&\mathsf{noninfluence} \equiv\\ &\forall\, d\ as\ bs\ s\ t.\\ &\quad \mathsf{reachable}\ s \wedge \mathsf{reachable}\ t \wedge s \overset{\mathsf{sources}\ as\ s\ d}{\approx} t \wedge s \overset{\mathsf{s\text{-}dom}}{\sim} t \wedge\\ &\quad \mathsf{ipurge}\ d\ as\ \{s\} = \mathsf{ipurge}\ d\ bs\ \{s\} \longrightarrow \mathsf{uwr\text{-}equiv}\ s\ as\ t\ bs\ d\end{aligned}$$

Note that, as a consequence of Lemma 1 introduced later, replacing the term $\mathsf{ipurge}\ d\ bs\ \{s\}$ by $\mathsf{ipurge}\ d\ bs\ \{t\}$ here yields an equivalent property.

noninfluence might be too strong a property for systems with a pre-determined static schedule that is fixed for the entire lifetime of the system and known to all domains. If every domain always knows the exact sequence of events that

must have gone before whenever it executes, then purging makes less sense. For these kinds of systems, an analogue of von Oheimb's weaker notion of *nonleakage* might be more appropriate. We denote this property nonleakage.

$$\mathsf{nonleakage} \equiv \forall\, as\ s\ t\ d.\ \mathsf{reachable}\ s \wedge \mathsf{reachable}\ t \wedge s \overset{\mathsf{s\text{-}dom}}{\sim} t \wedge s \overset{\mathsf{sources}\ as\ s\ d}{\approx} t \longrightarrow \mathsf{uwr\text{-}equiv}\ s\ as\ t\ as\ d$$

Naturally, noninfluence implies nonleakage.

2.4 Unwinding Conditions

A standard proof technique for noninterference properties involves proving so-called *unwinding conditions* [16] that examine individual execution steps of the system in question. We introduce two unwinding conditions. The first is sound and complete for nonleakage. The addition of the second to the first is sound and complete for noninfluence.

Both of these conditions examine individual execution steps of the system, and assert that they must all satisfy specific properties. As is usual with noninterference, we would like to conclude that these same properties are true across all runs of the system. However, this rests on the assumption that a run of the system, say in which it performs some sequence of actions *as*, is equivalent to performing a sequence of one-step executions for each of the events in *as* in turn.

This is formalised by the following function Run, which takes a step function *Stepf*, and repeatedly applies it to perform a sequence of actions *as* by executing each action in *as* in turn.

$$\mathsf{Run}\ Stepf\ [] = \{(s, s) \mid \mathsf{True}\}$$
$$\mathsf{Run}\ Stepf\ (a{\cdot}as) = Stepf\ a \circ \mathsf{Run}\ Stepf\ as$$

Like ours, traditional unwinding conditions are predicated on the assumption that $\mathsf{reachable}\ s \longrightarrow \mathsf{execution}\ A\ s\ as = \{s' \mid (s, s') \in \mathsf{Run\ Step}\ as\}$ (assuming naturally that $\mathsf{reachable}\ \mathsf{s}_0$ too). While this is valid for traditional noninterference formulations, in which their execution is defined exactly in this way [21], it is not always true for an arbitrary data-type automaton of the kind introduced in Section 2.1 over which our noninterference properties are defined. However, for most well behaved data types this condition is true, and certainly holds for those that model the seL4 functional specification and its C implementation. Thus we restrict our attention to those automata A that satisfy this assumption and return to our unwinding conditions.

The first unwinding condition is a confidentiality property, while the second is an integrity property. The confidentiality property we denote confidentiality-u, and resembles the conjunction of von Oheimb's *weak step consistency* and *step respect* [21] for deterministic systems; however, we require it to hold for *all* successor states and to take into account the scheduler domain s-dom.

$$\mathsf{confidentiality\text{-}u} \equiv \forall\, a\ d\ s\ t\ s'\ t'.\ \mathsf{reachable}\ s \wedge \mathsf{reachable}\ t \wedge s \overset{\mathsf{s\text{-}dom}}{\sim} t \wedge s \overset{d}{\sim} t \wedge (\mathsf{dom}\ a\ s \rightsquigarrow d \longrightarrow s \overset{\mathsf{dom}\ a\ s}{\sim} t) \wedge (s, s') \in \mathsf{Step}\ a \wedge (t, t') \in \mathsf{Step}\ a \longrightarrow s' \overset{d}{\sim} t'$$

This property says that the contents of a domain d after an action a occurs can depend only on d's contents before a occurred, as well as the contents of the domain $\mathsf{dom}\ a\ s$ performing a if that domain is allowed to send information to d. This condition alone allows d to perhaps infer that a has occurred, but not to learn anything about the contents of confidential domains.

The second unwinding condition is an integrity property, denoted integrity-u, and is essentially Rushby's *local respect* [16] adapted to nondeterministic systems and again asserted for all successor states.

$$\mathsf{integrity\text{-}u} \equiv \forall\, a\ d\ s\ s'.\ \mathsf{reachable}\ s \wedge \mathsf{dom}\ a\ s \not\rightsquigarrow d \wedge (s,\ s') \in \mathsf{Step}\ a \longrightarrow s \overset{d}{\sim} s'$$

It says that an action a that occurs from some state s can affect only those domains that the domain performing the action, $\mathsf{dom}\ a\ s$, is allowed to directly send information to. It prevents any domain d for which $\mathsf{dom}\ a\ s \not\rightsquigarrow d$ from even knowing that a has occurred.

The soundness proofs for these unwinding conditions are slightly more involved than traditional proofs of soundness for unwinding conditions. This is because our sources and ipurge functions are both state-dependent. The following lemma is useful for characterising those states that agree on sources and ipurge, under confidentiality-u, namely those related by $\overset{\mathsf{s\text{-}dom}}{\sim}$.

Lemma 1. $\mathsf{confidentiality\text{-}u} \wedge \mathsf{reachable}\ s \wedge \mathsf{reachable}\ t \wedge s \overset{\mathsf{s\text{-}dom}}{\sim} t \longrightarrow$
$\mathsf{sources}\ as\ s\ d = \mathsf{sources}\ as\ t\ d \wedge \mathsf{ipurge}\ d\ as\ \{s\} = \mathsf{ipurge}\ d\ as\ \{t\}$

With this result, the proofs of the soundness of our unwinding conditions are similar to those for traditional non-state-dependent formulations of noninterference, since (as we explain shortly) confidentiality-u guarantees that the equivalence $\overset{\mathsf{s\text{-}dom}}{\sim}$, asserted by noninfluence and nonleakage, is always maintained. The completeness proofs for these unwinding conditions are straightforward.

Theorem 1 (Soundness and Completeness).

$\mathsf{nonleakage} = \mathsf{confidentiality\text{-}u}$, *and* $\mathsf{noninfluence} = (\mathsf{confidentiality\text{-}u} \wedge \mathsf{integrity\text{-}u})$

2.5 Scheduling

We said that our noninterference formulation requires us to show that the scheduler's choices are independent of the other domains. To see why, consider when the domain d from our unwinding conditions is s-dom. Then confidentiality-u implies that s-dom can never be affected by the state of the other domains:

$$\forall\, a\ s\ t\ s'\ t'.\ \mathsf{reachable}\ s \wedge \mathsf{reachable}\ t \wedge s \overset{\mathsf{s\text{-}dom}}{\sim} t \wedge (s,\ s') \in \mathsf{Step}\ a \wedge (t,\ t') \in \mathsf{Step}\ a \longrightarrow s' \overset{\mathsf{s\text{-}dom}}{\sim} t'$$

Thus confidentiality-u implies that $\overset{\mathsf{s\text{-}dom}}{\sim}$ is always maintained.

When $\mathsf{dom}\ a\ s \neq \mathsf{s\text{-}dom}$, $\mathsf{dom}\ a\ s \not\rightsquigarrow \mathsf{s\text{-}dom}$. So integrity-u implies that the scheduler domain can never be affected by the actions of the other domains:

$$\forall\, a\ s\ s'.\ \mathsf{reachable}\ s \wedge \mathsf{dom}\ a\ s \neq \mathsf{s\text{-}dom} \wedge (s,\ s') \in \mathsf{Step}\ a \longrightarrow s \overset{\mathsf{s\text{-}dom}}{\sim} s'$$

2.6 Refinement

We now show that noninfluence and nonleakage are preserved by refinement. This means we can prove them of an abstract specification A and conclude that they hold for all concrete implementations C that refine it (i.e. for which $A \sqsubseteq C$).

Theorem 2 (noninfluence **and** nonleakage **are Refinement-Closed).**
When $A \sqsubseteq C$, $\mathsf{noninfluence}_A \longrightarrow \mathsf{noninfluence}_C$, *and* $\mathsf{nonleakage}_A \longrightarrow \mathsf{nonleakage}_C$

Proof. We will prove that each unwinding condition is closed under refinement, which implies that their conjunction is as well. The result then follows from Theorem 1. Let A and C be two automata, and write Step_A, $\mathsf{sources}_A$ etc. for those respective functions applied to A and similarly for C. Then, when $A \sqsubseteq C$, C's executions are a subset of A's, so $\mathsf{reachable}_C\ s \longrightarrow \mathsf{reachable}_A\ s$ and $\mathsf{Step}_C\ a \subseteq \mathsf{Step}_A\ a$. It is straightforward to show then that $\mathsf{integrity\text{-}u}_A \longrightarrow \mathsf{integrity\text{-}u}_C$ and $\mathsf{confidentiality\text{-}u}_A \longrightarrow \mathsf{confidentiality}\text{-}u_C$, as required. □

As mentioned earlier, a consequence of being preserved by refinement is that our unwinding conditions tolerate very little nondeterminism. Specifically, if the unwinding conditions hold, a system must have no *domain-visible* nondeterminism, which is nondeterminism that can be observed by any domain. This is because any such nondeterminism could abstract from a confidential source of information that is present in a refinement, and so implies the existence of insecure refinements. The following lemma states this restriction formally.

Lemma 2 (No Visible Nondeterminism).

$\mathsf{confidentiality\text{-}u} \wedge \mathsf{reachable}\ s \wedge (s, s') \in \mathsf{Step}\ a \wedge (s, s'') \in \mathsf{Step}\ a \longrightarrow s' \overset{d}{\sim} s''$

3 A Proof Calculus for Confidentiality for State Monads

Having explained our noninterference formulation, and in particular its unwinding conditions, we now present a proof calculus for discharging these unwinding conditions across an OS kernel. We have successfully applied this calculus to the seL4 microkernel, as part of ongoing work to prove that it enforces our noninterference formulation.

Our proof calculus operates over nondeterministic state monads, the formalism that underpins the seL4 abstract functional specification. Specifically, the internal steps of the automaton that embodies the seL4 functional specification are formalised as computations of a nondeterministic state monad. The state type of this monad is simply the internal state of the automaton, which for the seL4 functional specification is also identical to its observable state. The unwinding condition integrity-u asserts that the state before a single execution step is related to each final state after the execution step. It is naturally phrased as a Hoare triple, and discharged using standard verification techniques. For seL4, we have used a monadic Hoare logic and its associated verification condition generator (VCG) [6] to discharge this condition [17]. This leaves just the property confidentiality-u. It is this property that our confidentiality proof calculus addresses.

3.1 Nondeterministic State Monad

To prove confidentiality for an entire kernel specification, we need to be able to decompose it across that specification to make verification tractable. It is this challenge that our proof calculus addresses for nondeterministic state monads.

The type for this nondeterministic state monad is

$$state \Rightarrow (\alpha \times state)\ set \times bool$$

That is, it is a function that takes a state s as its sole argument, and returns a pair p. The first component $\mathsf{fst}\ p$ is a set of pairs $(rv,\ s')$, where rv is a return-value and s' is the result state. Each such pair $(rv,\ s')$ represents a possible execution of the monad. The presence of more than one element in this set implies that the execution is nondeterministic. The second part $\mathsf{snd}\ p$ of the pair returned by the monad is a boolean flag, indicating whether at least one of the computations has failed. Since our confidentiality property is termination insensitive, this flag can be ignored for the purpose of this paper.

Our proof calculus for confidentiality properties over this state monad builds upon the simpler proof calculus for Hoare triples [6] mentioned above. In this calculus, a precondition P is a function of type $state \Rightarrow bool$, i.e. a function P such that, a given state s satisfies P if and only if $P\ s$ is true. Since a monad f returns a set of return-value/result-state pairs, a postcondition Q is a function of type $\alpha \Rightarrow state \Rightarrow bool$. Q may be viewed as a function that given a return-value rv and corresponding result-state s', tells whether they meet some criteria. Alternatively, Q may be viewed as a function that, given some return-value rv, yields a state-predicate $Q\ rv$ that tests validity of the corresponding result-state s'. We write such Hoare triples as $\{\!|P|\!\}\ f\ \{\!|Q|\!\}$, and define their meaning as follows.

$$\{\!|P|\!\}\ f\ \{\!|Q|\!\} \equiv \forall s.\ P\ s \longrightarrow (\forall (rv,\ s') \in \mathsf{fst}\ (f\ s).\ Q\ rv\ s')$$

The proof calculus for Hoare triples of our nondeterministic state monad includes a mechanical rule application engine that acts as a VCG for discharging Hoare triples [6]. Later we discuss how we can apply this same engine to act as a VCG for discharging our confidentiality properties.

3.2 Confidentiality over State Monads

Observe that the property, confidentiality-u, addressed by our confidentiality proof calculus considers two *pre-states*, s and t, for which some equivalences hold, and then asserts that for all *post-states*, s' and t', another equivalence holds. We formalise this for our nondeterministic state monad, generalising over the pre- and post-state equivalences, as the property $\mathsf{ev}\ A\ B\ P\ f$, pronounced *equivalence valid*. Here, A and B are pre-state and post-state equivalence relations respectively (often called just the *pre-equivalence* and *post-equivalence* respectively), f is the monadic computation being executed and P is a precondition that the pre-states s and t are assumed to satisfy.

$$\mathsf{ev}\ A\ B\ P\ f \equiv$$
$$\forall s\ t.\ P\ s \wedge P\ t \wedge A\ s\ t \longrightarrow (\forall (r_a,\ s') \in \mathsf{fst}\ (f\ s).\ \forall (r_b,\ t') \in \mathsf{fst}\ (f\ t).\ r_a = r_b \wedge B\ s'\ t')$$

Note that $\mathsf{ev}\ A\ B\ P\ f$ also asserts that the return values from both executions of f are equal. This requires that these return-values be derived only from those parts of the system state that are identical between the two executions (i.e. those parts that the pre-equivalence A guarantees are identical). The purpose of the precondition P is to allow extra conditions under which the pre-equivalence A guarantees that confidentiality is satisfied. For instance, if f is a function that reads a region of user memory, the precondition P might include a condition that ensures that this region is covered by the pre-equivalence A.

To decompose this property across a monadic specification, we need to define proof rules for the basic monad operators, return and $\gg=$ (pronounced "bind"). $\mathsf{return}\ x$ is the state monad that leaves the state unchanged and simply returns the value x. This means that if A holds for the pre-states, then A will hold for the post-states as well. Also, $\mathsf{return}\ x$ always returns the same value (x) when called. This gives us the following proof rule.

$$\frac{}{\mathsf{ev}\ A\ A\ (\lambda\text{-}.\ \mathsf{True})\ (\mathsf{return}\ x)}\ \text{RETURN-EV}$$

Note that this rule restricts the post-equivalence to be the same as the pre-equivalence. As we explain shortly, this turns out not to be a problem in practice.

$f \gg= g$ is the composite computation that runs f, and then runs g on the result, and is used to sequence computations together. Specifically, $f \gg= g$ runs the computation f to obtain a return value rv and result state s', and then calls $g\ rv$ to obtain a second computation that is run on the state s'. Because f might be nondeterministic, $f \gg= g$ does this for all pairs $(rv,\ s')$ that f emits, taking the distributed union over all results returned from each $g\ rv\ s'$.

Because we want to be able to decompose the proof of ev across a specification, we need a proof rule for $f \gg= g$ that allows us to prove ev for f and g separately, and then combine the results to obtain a result overall. The following proof rule, BIND-EV, does exactly that.

$$\frac{\forall\, rv.\ \mathsf{ev}\ B\ C\ (Q\ rv)\ (g\ rv) \qquad \mathsf{ev}\ A\ B\ P'\ f \qquad \{\!|P''|\!\}\ f\ \{\!|Q|\!\}}{\mathsf{ev}\ A\ C\ (P'\ \mathsf{and}\ P'')\ (f \gg= g)}\ \text{BIND-EV}$$

Here, $P'\ \mathsf{and}\ P''$ is the conjunction of preconditions P' and P'', i.e. $P'\ \mathsf{and}\ P'' \equiv \lambda s.\ P'\ s \wedge P''\ s$. BIND-EV can be read as a recipe for finding a precondition $?P$ such that $\mathsf{ev}\ A\ C\ ?P\ (f \gg= g)$ is true. First, for any return value rv that f might emit, find some state-equivalence B and a precondition $Q\ rv$, which may mention rv, such that $g\ rv$ yields post-states that satisfy the post-equivalence C. Secondly, find a precondition P' such that executing f yields post-states that satisfy the just found state-equivalence B. Finally, find a precondition P'', such that for all return-values rv emitted from executing f, their corresponding result-states satisfy $Q\ rv$. The desired precondition $?P$ is then $P'\ \mathsf{and}\ P''$.

This rule works because if ev is true for f, we know that both executions of f yield the same return-value, say rv, which means that the two subsequent executions both run the same computation, $g\ rv$. In the rare case that ev cannot be proved for f (say because f returns a value rv derived from confidential state), a more sophisticated rule is required that we introduce later in Section 3.4.

3.3 Automating Confidentiality Proofs

Note that when $C = A$, we may define a simpler variant of BIND-EV, called BIND-EV', in which B and C are both A.

$$\frac{\forall rv.\ \mathsf{ev}\ A\ A\ (Q\ rv)\ (g\ rv) \qquad \mathsf{ev}\ A\ A\ P'\ f \qquad \{\!|P''|\!\}\ f\ \{\!|Q|\!\}}{\mathsf{ev}\ A\ A\ (P'\ \mathsf{and}\ P'')\ (f \gg\!= g)}\ \text{BIND-EV'}$$

To apply this rule, we need only compute sufficient preconditions $Q\ rv$, P' and P'' for the relevant obligations. Our ordinary Hoare logic VCG can be applied to compute P'', of course, while BIND-EV' is itself a recipe for computing sufficient $Q\ rv$ and P'. In other words, we may recursively apply BIND-EV' to compute appropriate $Q\ rv$ and P', given appropriate ev rules for the primitive monadic functions.

This is precisely the technique that we have taken to prove these statements across the seL4 functional specification. Specifically, at the top-level, the pre-equivalence of confidentiality-u, asserted for s and t, implies the post-equivalence, $\overset{d}{\sim}$, asserted for all s' and t', because the pre-equivalence includes $\overset{d}{\sim}$. Hence, if we prove that the pre-equivalence is preserved, we can deduce that the post-equivalence must hold after each kernel event. This allows us to reason about a more restricted version of ev in which the pre- and post-equivalences are always identical, using rules like RETURN-EV and BIND-EV' above.

The rule-application engine developed previously [6] that acts as a VCG for Hoare triples over our nondeterministic state monad, can be applied directly without any modification to discharge ev statements by feeding it the appropriate rules. It requires rules like BIND-EV', to decompose these proofs into smaller goals, as well as appropriate rules, like RETURN-EV, to discharge the goals at the leaves of the proof tree. Familiar rules from prior work on proof methods for relational properties of programs [1, 2, 4] may be derived for other monadic functions, such as the one below for monadic while-loops. It establishes confidentiality for the loop under the invariant P when the loop body B maintains confidentiality and the pre-equivalence A guarantees that both executions terminate together. The loop body B and condition C are both parametrised by a loop parameter n, which for subsequent loop iterations is the return-value of the previous iteration.

$$\frac{\begin{array}{c}\forall s\ t\ n.\ A\ s\ t \wedge P\ n\ s \wedge P\ n\ t \longrightarrow C\ n\ s = C\ n\ t \\ \forall n.\ \{\!|P\ n\ \mathsf{and}\ C\ n|\!\}\ B\ n\ \{\!|P|\!\} \qquad \forall n.\ \mathsf{ev}\ A\ A\ (P\ n\ \mathsf{and}\ C\ n)\ (B\ n)\end{array}}{\mathsf{ev}\ A\ A\ (P\ n)\ (\mathsf{whileLoop}\ C\ B\ n)}\ \text{WHILE-EV}$$

3.4 Proving the Functions that Read Confidential State

The approach so far allows very automatic proofs for functions that do not read any confidential state, and so always yield identical return-values rv. Because these functions make up the bulk of seL4, this is what our calculus has been tuned for. However, it is less well suited to functions that operate on confidential state without revealing it to unauthorised domains. Our approach requires confidentiality proofs for these kinds of functions to be performed more manually.

$$\frac{\forall s\ t.\ P\ s \wedge P'\ t \wedge A\ s\ t \longrightarrow R\ x\ y}{\mathsf{ev2}\ A\ A\ R\ P\ P'\ (\mathsf{return}\ x)\ (\mathsf{return}\ y)}\ \textsc{return-ev2}$$

$$\frac{\begin{array}{c}\forall rv\ rv'.\ R'\ rv\ rv' \longrightarrow \mathsf{ev2}\ B\ C\ R\ (Q\ rv)\ (Q'\ rv')\ (g\ rv)\ (g'\ rv') \\ \mathsf{ev2}\ A\ B\ R'\ P\ P'\ f\ f' \qquad \{\!|S|\!\}\ f\ \{\!|Q|\!\} \qquad \{\!|S'|\!\}\ f'\ \{\!|Q'|\!\}\end{array}}{\mathsf{ev2}\ A\ C\ R\ (P\ \mathsf{and}\ S)\ (P'\ \mathsf{and}\ S')\ (f \gg\!= g)\ (f' \gg\!= g')}\ \textsc{bind-ev2}$$

Fig. 1. VCG rules for ev2

An example is the seL4 function send-async-ipc, which sends a message on an *asynchronous endpoint*. Asynchronous endpoints facilitate unidirectional communication, which implies that the act of sending on an asynchronous endpoint should not leak any information back to the sender. Sending such a message does require the kernel to read state outside of the sending domain (such as state in the endpoint); however, it should not reveal any of this state to the sender.

There is no guarantee, then, that when the two executions of send-async-ipc that ev compares each read the internal state of the endpoint in question, they will get the same result. This means their subsequent executions might behave differently to each other. Proving that ev holds in this case requires comparing two different executions, and showing that they establish the post-equivalence. This suggests that we should reason about a more general property than ev that can talk about two different executions, and allows return-values to differ.

These insights lead to the following property, called ev2.

$$\begin{array}{l}\mathsf{ev2}\ A\ B\ R\ P\ P'\ f\ f' \equiv \\ \forall s\ t.\ P\ s \wedge P'\ t \wedge A\ s\ t \longrightarrow \\ \quad (\forall (r_a,\ s') \in \mathsf{fst}\ (f\ s).\ \forall (r_b,\ t') \in \mathsf{fst}\ (f'\ t).\ R\ r_a\ r_b \wedge B\ s'\ t')\end{array}$$

ev2 takes two computations, f and f', and two associated preconditions, P and P'. It also takes a return-value relation R, that it asserts holds for the return-values of f and f'. ev2 generalises ev, specifically $\mathsf{ev}\ A\ B\ P\ f \equiv \mathsf{ev2}\ A\ B\ \mathsf{op} =\ P\ P\ f\ f$, where op = is the equality operator. One usually applies this equivalence to rewrite ev goals that cannot be proved by the VCG, into ev2 goals. One then manually applies proof rules like in Figure 1 to discharge these goals.

Applying BIND-EV2 usually requires the human to come up with an appropriate intermediate return-value relation R' that will hold for the return values emitted from f and f'. As with ev, we usually work with a simpler rule in which $(B = C) = A$, which we omit for brevity. We suspect that techniques could be borrowed from other work on automatically proving confidentiality properties of programs [2, 18] to help automatically infer appropriate R'. However, because ev2 proofs are seldom required for seL4, we have not needed to implement them.

4 Related Work

Recently, Barthe et al. [3] presented a formalisation of isolation for an *idealised* model of a hypervisor, and its unwinding conditions. Like ours, their definition

is based on von Oheimb's noninfluence [21]. As in traditional formalisations of noninterference, in their formulation actions are intrinsically linked to domains, and so it cannot reason about information leaks through scheduling decisions.

INTEGRITY-178B is a real-time operating system for which an isolation proof has been completed [15]. The isolation property proved is based on the GWVr2 information flow property [9], which bears similarities to the unwinding conditions for noninterference. Like ours, it is general enough to handle systems in which previous execution steps affect which is the entity that executes next. Unlike ours, it is defined only for deterministic systems. The exact relationship between GWVr2 and our conditions deserves further study.

Our formulation of information flow security is descendant from traditional *ipurge*-based formulations of intransitive noninterference (starting with Haigh and Young's [10]). Van der Meyden [19] argues that ipurge-based formulations of intransitive noninterference are too weak for certain intransitive policies, and proposes a number of stronger definitions. He shows that Rushby's unwinding conditions [16] are sufficient for some of these alternatives. Given the similarity of our unwinding conditions to Rushby's, we wonder whether our existing unwinding conditions may be sufficient to prove analogues of van der Meyden's definitions.

Others have presented noninterference conditions for systems with scheduling components. One recent example is van der Meyden and Zhang [20], who consider systems that run in lock-step with a scheduling component that controls which domain's actions are currently enabled. Their security condition for the scheduler requires that the actions of the High domain cannot affect scheduling decisions. Our formulation, in contrast, has the scheduler update a component of the system state that determines the currently running domain. This allows our scheduler security condition to require that scheduling decisions be unaffected not only by domain actions, but also by domain state.

A range of proof calculi and verification procedures for confidentiality properties, and other *relational* properties, have also been developed [1,2,4,5,18]. Unlike many of these, ours aims not at generality but rather at scalability. The simplicity of our calculus has enabled it to scale to the entire functional specification of the seL4 microkernel, whose size is around 2,500 lines of Isabelle/HOL, and whose implementation that refines this specification is around 8,500 lines of C.

5 Conclusion

We have presented a definition of noninterference for operating system kernels, with sound and complete unwinding conditions. We have shown how these latter can be implemented in a proof calculus for nondeterministic state monads with automation support. Our success in applying both of these to the seL4 microkernel, in an ongoing effort to prove that it enforces noninterference, attest to their practical utility and applicability to programs on the order of 10,000 lines of C.

Acknowledgements We thank Kai Engelhardt, Sean Seefried, and Timothy Bourke for their comments on earlier drafts of this paper.

References

1. T. Amtoft and A. Banerjee. Information flow analysis in logical form. In *SAS '04*, volume 3148 of *LNCS*, pages 33–36. Springer-Verlag, 2004.
2. T. Amtoft and A. Banerjee. Verification condition generation for conditional information flow. In *FMSE '07*, pages 2–11. ACM, 2007.
3. G. Barthe, G. Betarte, J. Campo, and C. Luna. Formally verifying isolation and availability in an idealized model of virtualization. In M. Butler and W. Schulte, editors, *17th FM*, volume 6664 of *LNCS*, pages 231–245. Springer-Verlag, 2011.
4. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL 2004*, pages 14–25. ACM, 2004.
5. L. Beringer. Relational decomposition. In *2nd ITP*, volume 6898 of *LNCS*, pages 39–54. Springer-Verlag, 2011.
6. D. Cock, G. Klein, and T. Sewell. Secure microkernels, state monads and scalable refinement. In *21st TPHOLs*, volume 5170 of *LNCS*, pages 167–182, Aug 2008.
7. W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.
8. J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symp. Security & Privacy*, pages 11–20, Oakland, California, USA, Apr 1982. IEEE.
9. D. A. Greve. Information security modeling and analysis. In D. S. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 249–300. Springer-Verlag, 2010.
10. J. T. Haigh and W. D. Young. Extending the noninterference version of MLS for SAT. *Trans. Softw. Engin.*, 13:141–150, Feb 1987.
11. 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. ACM, 2009.
12. G. Klein, T. Murray, P. Gammie, T. Sewell, and S. Winwood. Provable security: How feasible is it? In *13th HotOS*, pages 28–32, Napa, CA, USA, May 2011. USENIX.
13. D. Matichuk and T. Murray. Extensible specifications for automatic re-use of specifications and proofs. In *10th SEFM*, Oct 2012.
14. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
15. R. J. Richards. Modeling and security analysis of a commercial real-time operating system kernel. In D. S. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 301–322. Springer-Verlag, 2010.
16. J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, SRI International, Dec 1992.
17. T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein. seL4 enforces integrity. In *2nd ITP*, volume 6898 of *LNCS*, pages 325–340, Nijmegen, The Netherlands, Aug 2011. Springer-Verlag.
18. T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *SAS '05*, volume 3672 of *LNCS*, pages 352–367. Springer-Verlag, 2005.
19. R. van der Meyden. What, indeed, is intransitive noninterference? In *12th ESORICS*, volume 4734 of *LNCS*, pages 235–250. Springer-Verlag, 2007.
20. R. van der Meyden and C. Zhang. Information flow in systems with schedulers. In *21st CSF*, pages 301–312. IEEE, Jun 2008.
21. D. von Oheimb. Information flow control revisited: Noninfluence = noninterference + nonleakage. In *9th ESORICS*, volume 3193 of *LNCS*, pages 225–243, 2004.