Secure architectures on a verified microkernel

Andrew Boyton

School of Computer Science and Engineering

Univeristy of New South Wales

Sydney, Australia

Submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

August 2014

THE UNIVERSITY OF NEW SOUTH WALES
Thesis/Dissertation Sheet

Surname or Family name: Boyton

First name: Andrew **Other name/s:** James

Abbreviation for degree as given in the University calendar: PhD

School: Computer Science and Engineering **Faculty:** Engineering

Title: Secure architectures on a verified microkernel

Abstract 350 words maximum:

The safety and security of software systems depends on how they are initially configured. Manually writing program code that establishes such an initial configuration is a tedious and error-prone process, and yet most systems now are initialised with manually written, ad-hoc code.

This thesis provides a solution to this process, presenting an automatic and formally verified system initialiser for component-based systems built on the general-purpose microkernel seL4.

The initialiser takes a declarative formal description of the desired initialised state, and uses seL4-provided services to create all necessary components, setup their communication channels, and distribute the required capabilities.

We analyse a model for capability-based systems, namely the take-grant protection model, and extend the existing literature to develop a formal model in Isabelle/HOL that models real-world capability-based systems such as seL4 more accurately. We use this to demonstrate how the security of a system can be conferred by capabilities.

We provide a formal algorithm of system initialisation and prove, in the theorem prover Isabelle/HOL, that the resulting state conforms with the desired one, giving us an unprecedented level of assurance for the correctness of system initialisation. Our proof formally connects to the existing functional correctness proof of the seL4 microkernel.

In the process of this work, we develop a custom separation algebra, with a fine-level of granularity, for reasoning about both the API of the seL4 microkernel and the user-level code running on seL4.

FOR OFFICE USE ONLY Date of completion of requirements for Award:

Originality Statement

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation, and linguistic expression is acknowledged.'

Signed: Andrew Boyton

Date: 29/8/2014

Copyright Statement

Signed: Andrew Boyton

Date: 29/8/2014

Authenticity Statement

Signed: Andrew Boyton

Date: 29/8/2014

Abstract

The safety and security of software systems depends on how they are initially configured. Manually writing program code that establishes such an initial configuration is a tedious and error-prone process, and yet most systems now are initialised with manually written, ad-hoc code.

This thesis provides a solution to this process, presenting an automatic and formally verified system initialiser for component-based systems built on the general-purpose microkernel seL4.

The initialiser takes a declarative formal description of the desired initialised state, and uses seL4-provided services to create all necessary components, setup their communication channels, and distribute the required capabilities.

We analyse a model for capability-based systems, namely the take-grant protection model, and extend the existing literature to develop a formal model in Isabelle/HOL that models real-world capability-based systems such as seL4 more accurately. We use this to demonstrate how the security of a system can be conferred by capabilities.

We provide a formal algorithm of system initialisation and prove, in the theorem prover Isabelle/HOL, that the resulting state conforms with the desired one, giving us an unprecedented level of assurance for the correctness of system initialisation. Our proof formally connects to the existing functional correctness proof of the seL4 microkernel.

In the process of this work, we develop a custom separation algebra, with a fine-level of granularity, for reasoning about both the API of the seL4 microkernel and the user-level code running on seL4.

Acknowledgements

I would like to thank my supervisors, Gerwin Klein for his support and direction, and understanding of all things verification related, June Andronick for always helping me stay on track and work out how to explain things more clearly, and Kevin Elphinstone for his help with all things kernel related and for keeping a verification student a little more honest.

I would also like to thank all the people in the Trustworthy Systems group at NICTA. Your friendship and support has made doing this research rather enjoyable. In particular, I would like to thank David Greenaway, a dear brother in Christ and a constant help in all things PhD related, Rafal Kolanski for both introducing me to the wonderful world of formal verification at NICTA and for getting me started on separation logic in the first place. If only I had listened to you earlier my PhD might have been done sooner. I would like to thank Callum Bannister and Xin Gao for their help with formalising the seL4 API and with all things separation logic related, Thomas Sewell for all his frequent Isabelle help, and Matthew Fernandez for his advice with capDL and making C programs actually run on seL4.

Finally, I'd also like to thank my church family at Unichurch, members both past and present, for their love, friendship, and encouragement over the last decade or so, and my wonderful wife Chloe for putting up with me through my PhD and for her unconditional love, encouragement, and support throughout it all.

Contents

List of Figures

Chapter 1

Introduction

Complex computer systems are given increasing control in our lives. In many of these systems, a malfunction in a single part can take down the entire system.

One commonly used technique for mitigating the impact of malfunctioning software is to split larger systems into several smaller *components*, each with a limited level of control over the system. For example, Apple sandboxes applications, and allows the user to restrict games from accessing their phone's location (*iOS 7: Understanding Location Services* 2014). Google restricts the rendering agent in the Chrome browser from having access to the hard drive of your computer (Barth et al., 2008). File and web servers are often not run on a single operating system, but are separated into virtualised servers running on a single computer system to restrict their access to each other. We can achieve this sandboxing of applications by breaking systems into components, and putting these components in virtual "boxes" with reduced access and well-defined communication channels between them.

This idea is an old one. The *principle of least privilege*—giving components only the access that they require—was first coined by Saltzer (1974) to describe the design philosophy of the Multics operating system. The idea of only handing over the required access or information is of course much older—the military concept of "need-to-know" embodies this principle.

Rushby, examining the state of operating system security, proposed the concept of a *separation kernel* (Rushby, 1981), where software is split into several independent *components*. Individual components running on the separation kernel should run equivalently to as if they were running as a distributed system on discrete hardware, where the only mechanism for components to communicate or interact with each other is through explicit channels placed by the system designer. For example Figure 1.1 shows a simple system where an application is able to send information to a network only through a trusted encryption service.

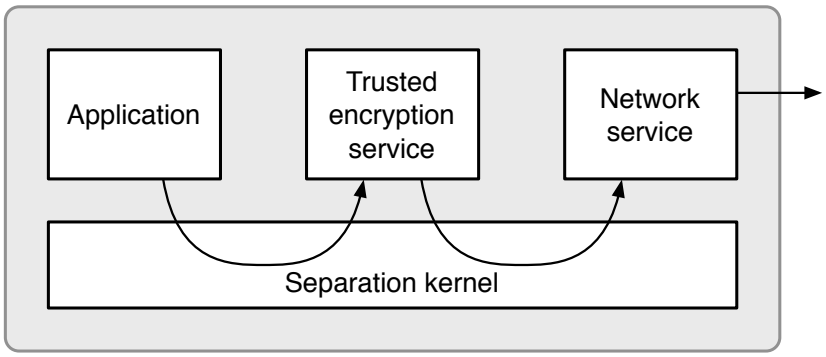


Figure 1.1: An example MILS-based system.

Systems with such a design are often said to implement the MILS philosophy—Multiple Independent Levels of Security (Alves-Foss et al., 2006). There are numerous MILS-style systems, such as Greenhills Integrity kernel (Greenhills Software, Inc., 2008), EROS (Shapiro, 1999), and seL4 (Klein et al., 2014). The one we concentrate on in this thesis is seL4, an open-source microkernel with a full functional correctness proof showing its C code correctly implements its abstract specification (Klein et al., 2009) and its compiled binary correctly implements the C source semantics (Sewell et al., 2013). On top of the abstract specification Sewell et al. (2011) showed that seL4 maintains authority confinement and integrity, and Murray et al. (2013) showed noninterference for the seL4 kernel. These results place an upper bound on how authority and information may flow in a seL4 system, based on the authority present in the capabilities held by the components of the system.

seL4 is a capability-based microkernel, meaning that authority is conferred by capabilities. Capabilities, first introduced by Dennis and Van Horn (1966), are an unforgeable, explicit set of access rights to an entity, such as a page in memory or a communication channel between components. seL4, being a take-grant based system (a concept we examine in detail in Chapter 4), uses capabilities to both confer the authority to communicate and confer the authority to transfer capabilities.

In this work, we are not interested in how to design or verify a MILS-based system, but instead examine the question "how do we correctly *initialise* such systems?". This question is especially important in MILS-based systems as the security of such a system generally requires the access control primitives to be correctly provisioned. For example, the authority confinement, integrity, and non-interference proofs for the seL4 microkernel all rely on the capabilities of the system being configured in conformance with the access control policy (Sewell et al., 2011; Murray et al., 2013). Without this, these security results are null and void.

The initialisation and configuration of computer systems is a hard problem. The abstractions commonly used to configure systems are generally in the process of being initialised themselves and thus cannot be used. Additionally, the initial processes that configure a system are often given elevated authority and special care must be taken to ensure that these processes appropriately manage and diminish this authority (Shapiro and Hardy, 2002).

Previous work in the correct initialisation of MILS-style systems has either been informal, or at best, modelled by high-level specifications which are disconnected from the operating system code. In this thesis, we move beyond the existing work of simply initialising MILS-style systems, and examine how we can do so both automatically and provably correctly.

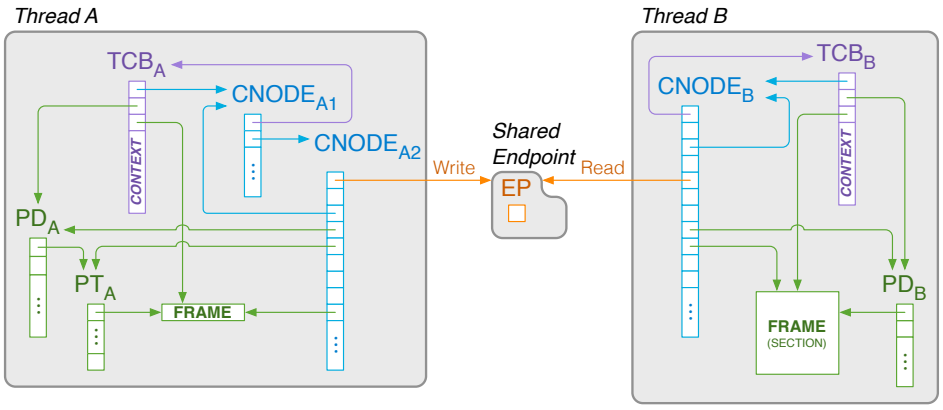


Figure 1.2: A very simple seL4-based system with two threads communicating via a shared endpoint.

A simple example of a system that we could initialise is illustrated in Figure 1.2. For seL4, we can specify the state of such a system using the capability distribution language (capDL), a declarative specification of the objects and capabilities within a system. CapDL is designed to have the right level of granularity for describing systems for initialisation, and is suitable for reasoning about the security of our system as it has been formalised and linked to the above authority confinement, integrity, and noninterference proofs of seL4 (Boyton et al., 2013).

In this thesis we propose starting our systems using a system initialiser—a fixed formal algorithm for the first task that runs on a seL4 system. Our system initialiser takes a declarative capDL specification and takes us from an initial seL4 boot state to a final initialised state in conformance with our given specification, as shown in Figure 1.3. We also prove the correctness of our algorithm with respect to a verified API for the seL4 microkernel.

We reason that each step of formalised algorithm brings us one step closer to an initialised system, and that after our system initialiser has run, the final state of the system will be in conformance with our given specification.

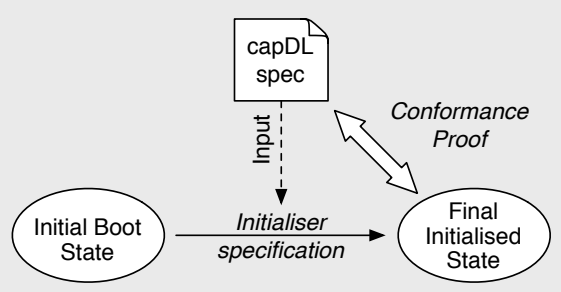


Figure 1.3: An overview of the system initialiser.

When showing that a given step gets us closed to an initialised system, we also need to ensure that this step is not destroying the work of preceding steps. In a running system, this is usually expressed as preserving a global invariant. The challenge when reasoning about an initialiser is that no invariant holds yet—the initialiser is in the process of creating a state that would satisfy the invariant of the running system. So for each step, we need to explicitly state (and prove) not only what each step of initialisation does but also what it does *not* modify. Standard Hoare logic makes such reasoning very hard-going. One approach that has been used to simplify such reasoning in the past is *separation logic*, first proposed by Reynolds (2002), which allows specifications to be written that both how memory changes and how memory remains unchanged. Because of this, we developed a separation logic for reasoning about the transformation of objects, and used this separation logic to prove the correctness of our system initialiser.

All of the definitions, theorems and proofs shown in this thesis have been verified using the theorem prover Isabelle/HOL, are presented in this document using Isabelle/HOL, and are available under an open-source licence (Trustworthy Systems Team, 2014).

1.1 Thesis objectives and contributions

The aim of this work is to be able to initialise a system in conformance with a given declarative specification of the desired system state, along with proving a formal proof that our final system state is correct. To achieve this, this thesis makes the following contributions:

- We develop a *formal algorithm* for system initialisation that takes a declarative specification of what we want our system to look like, using the capDL language, and transforms the state of seL4 from its initial boot state to a final state conforming to the given capDL specification. This process is illustrated in Figure 1.3.

- We develop a formal proof of the correctness of this system initialisation algorithm, proving that each step of our algorithm correctly creates and configures the objects described, while not undoing the work done by the previous steps. This proof is built on top of a verified specification of the kernel API for a real operating system kernel, seL4, with all of the intricacies that a real API specification entails.

- We prove the correctness of our algorithm using a separation logic that we develop. We build our separation logic on an abstract separation logic (Klein et al., 2012), which we have extended to allow reasoning about monadic maps and folding separation logic predicates over lists and sets. The separation logic we develop for this work is useful for reasoning about the transformation of objects at a sub-object level of granularity. Additionally, we show that separation logic, a logical framework commonly used to reason about pointer programs and concurrency is good fit for our problem domain.

- We develop a formalised take-grant protection model that extends the existing literature with the addition of shared capability storage as a first class citizen, and an explicit *create* right to be able to create an entity using a particular identifier (the latter of which allows entity destruction without the problematic entity identifier reuse issues of earlier work (Boyton, 2009; Shapiro, 1999)). This model is designed as a conceptual, high-level model for the seL4 microkernel, showing authority confinement and simple information flow results. This work predates that of Sewell et al. (2011) and Murray et al. (2013), and is not formally connected to the seL4 code itself, but is used to explore the implications of capability-based systems more broadly.

While the authority confinement, integrity, and non-interference proofs of Sewell et al. (2011) and Murray et al. (2013) rely on the state of a seL4 system to be in conformance with an access control policy, they do not provide any mechanism to ensure that this is the case.

Our work fills in this gap by providing a formal algorithm for a seL4 program that transforms the state of a seL4 system from its initial boot state into a final initialised state that is in conformance with a given capDL specification. As Boyton et al. (2013) has linked capDL to the above access control policies, our work allows us to have confidence that we can start a system correctly and that seL4 will enforce the security of our system as specified.

1.2 Outline

In this thesis we present our formalisation and proof of system initialisation, and show why this is important for establishing the security of an overall system. The chapters in this thesis are as follows:

Related work In Chapter 2 we examine the related work, examining how MILS-style systems are traditionally initialised, examining the related work that has been done using separation logic, and different formalisations of operating system security using the take-grant protection model.

Background In Chapter 3 we give a brief introduction to formal verification, Isabelle/HOL, separation logic, and seL4. We explore how it is we can have confidence in the correctness of our programs, and introduce the theorem prover Isabelle/HOL. We explain the notation of the theorem prover Isabelle/HOL, which mirrors that of mathematics, lambda calculus, and functional programming. We give a brief introduction to the style of reasoning that we employ in this thesis—separation logic, an extension of Hoare Logic. Finally, we conclude this chapter with an introduction to the seL4 microkernel (on which this work is based), the capDL language developed to describe protection states, and some of the existing body of proof that has developed for seL4. We concentrate on the authority confinement and integrity proofs (Sewell et al., 2011), and the non-interference proofs (Murray et al., 2013) developed for seL4, which show how we can use seL4 to build secure systems, assuming they are correctly configured.

Capability-based access control In Chapter 4 we examine the take-grant protection model, the security model on which seL4 kernel has been based. We present a formalisation of the take-grant protection model in Isabelle/HOL that extends the existing literature by adding shared capability storage as a first class citizen and an explicit create right to be able to create an entity using a particular identifier.

System initialisation In Chapter 5 we describe how component-based computer systems are often initialised, focusing on the capability-based seL4 microkernel. In this chapter we develop a formal algorithm for initialising seL4-based systems, taking a capDL specification and starting the system in conformance with this specification. We explain both the algorithm we developed for this thesis, which we formalise in Isabelle/HOL, as well as a C implementation. The C implementation was implemented in conjunction with others.

Separation logic In Chapter 6 we give an overview of separation logic, the algebraic structure behind separation logics, and the specific separation logic developed for this thesis. In this thesis, we build on an abstract separation logic

developed in conjunction with this thesis, as published in Klein et al. (2012). We show some extensions to this abstract separation logic that we developed for this thesis.

Correctness of system initialisation In Chapter 7 we describe the formal correctness result of the system initialisation algorithm we developed, how we proved it, and how we joined it to the existing seL4 proofs. In Section 7.4 we discuss the seL4 API specifications developed by others alongside this thesis and how it builds on the separation logic we developed in Chapter 6.

Conclusion We conclude in Chapter 8 with a discussion of our experience, limitations, and future work.

We begin in the next chapter by looking at existing work our own work builds upon, and work that is related to our own.

Chapter 2

Related work

2.1 Operating System Initialisation

Initialising systems according to a given configuration and guaranteeing that the initialisation is correct are both hard and critical tasks. We examine here how various security-focused operating systems are initialised, and what assurances they provide to ensure the correctness of their initialisation.

Separation kernels Security requirements for high-assurance certification of separation kernels, for instance, include providing evidence that the initialisation function establishes the system in a secure state consistent with the configuration data (National Security Agency, 2007). Configuration data describes high-level partitions and authorised information flows between partitions. In existing systems, this the correctness of initialisation is at best modelled by high-level specifications—the formal models of the MASK microkernel (Martin et al., 2002) and the formal specification of PikeOS (Verbeek et al., 2014) both assume an initial state conforming to the specified configuration exists. Our work assumes the kernel has correctly initialised a single root task, and we then reason about the correctness of this root task that initialises the rest of the system.

SELinux Rather than building a secure operating system from scratch, SELinux instead grafts security into Linux by adding a reference monitor to authorise all security sensitive operations according to an security policy. These SELinux policies allow fine grained MLS security, but the richness of these policies makes it difficult to understand them. Hicks et al. (2007) developed formal semantics for SELinux policies in Prolog and demonstrated that it was possible to show information flow properties of SELinux policies, but this model has not been formally connected to the SELinux code. For seL4, we can not only analyse the information flow properties of our policy, but also know that seL4 enforces these policies (Murray

et al., 2013). The question with seL4 remains on how to initialise our system in conformance with this policy.

L4 based systems In L4 based systems, such as seL4, the first task that runs is generally responsible for the initialisation of components at startup time. This task is usually large, for example, in Nizza-based systems, the size of the loader that starts applications is 37,000 lines of code, which is substantially larger than the L4 kernel itself (Singaravelu et al., 2006). Similarly, a simple case study on seL4 for a secure access controller (Andronick et al., 2010) had a trusted router manager that started up and tore down Linux virtual machines that was approximately 1,500 lines of code, which, whilst substantially smaller than the loader of Nizza, was custom built to only start Linux. A high-level security model of the trusted router manager was built based on an early version of the take-grant protection model we developed for this thesis and present in Chapter 4. This model was shown to have the desired security properties, although verification of the implementation was never attempted. Our work not only models the code for initialising systems at a much lower level of abstraction, but we prove the correctness of this formal algorithm of system initialisation by formally joining it to the seL4 kernel model.

EROS Microkernel The EROS microkernel (Shapiro, 1999) partially side-steps the initialisation problem by persistence; it simply restarts at the last saved checkpoint. The initial system image is constructed by hand (Shapiro and Hardy, 2002) and the creation and instantiation of confined subsystems uses *constructors* that are part of the trusted computing base (Shapiro and Weber, 2000). The proof of the correctness of these constructors is with respect to a high-level model of EROS, not formally linked to the EROS code. What sets our work apart is the proof of correctness of our model is shown by joining to the seL4 code itself.

OKL4 microkernel The OKL4 microkernel (Open Kernel Labs, 2008b) moves the initialisation problem almost entirely to offline processing and runs the initialisation phase once, before the system image is built using a process called *Elfweaving* (Open Kernel Labs, 2008a). The Elfweaving is done using a program written in Python. Similarly to EROS, when the machine starts, it loads a fully pre-initialised state. While this makes it possible to inspect the initialised state offline, a full assurance case must be made for each system, which is something we avoid by proving the correctness of our initialisation algorithm.

2.2 Separation Logic

In this work we construct a custom separation logic for reasoning about both a formalised seL4 API and the correctness of the system initialiser algorithm. In this

section we give an overview of the existing literature for separation logic, and how our work fits in to their body of work.

Separation algebras The foundations and algebraic structure of separation logic have been analysed by Calcagno et al. (2007), Dang et al. (2011), and Dockins et al. (2009). Many formalisations of separation logics base their definitions on the axioms of one of these, however the logics they construct are generally not reusable by other projects. To address this, Klein et al. (2012) developed an abstract separation logic, based on the abstract separation algebra of Calcagno et al. (2007), and instantiated a number of separation logics on top of this abstract separation logic, including the separation logic presented in this work.[1] This abstract separation logic of Klein et al. (2012) was developed in conjunction with this work. Our work uses the abstract separation logic developed by Klein et al. (2012) to construct the separation logic that we present in this work.

Permissional models Separation logic is commonly used for reasoning about concurrent programs where multiple threads can read and/or write shared data structures. To cope with this, Bornat et al. (2005), Parkinson (2005), and Dockins et al. (2009) have introduced different permission models for separation logic. Since our system initialiser is single threaded and is the only thread running in the system, we have avoided the complexities that these permission models bring. In this work we specify the state that an operation reads (but does not modify) by specifying it in the pre and postconditions of a Hoare triple. This gives us a simpler model to reason about.

Heap models The Burstall-Bornat model (developed by Burstall (1972) and formalised by Bornat (2000)) is commonly used to reason about memory heaps with different types of objects (Tuch, 2008; Filliâtre and Marché, 2007). In this model, the object heap is modelled as a collection of heaps. In our approach we use a variation of this model, mapping the multiple heaps of the Burstall-Bornat model to a single component heap from object identifiers, combined with component identifiers, to components of an object, where each object contains the *same* types of components.[2] This construction is described in detail in Section 6.5. This modification allows both a more convenient definition of a separation logic for our purposes, as well as more convenient reasoning using the logic.

[1] Klein et al. (2012) presented an abstract separation logic, and a number of instantiations of this logic, including one that we developed for this work. The version we presented in Klein et al. (2012) has a different heap model to the heap model we present here, as we explain in Section 6.5.

[2] In our model the similarity between different types of objects is much greater than their differences, making treating their components (namely the capabilities and the fields of an object) as the same type more convenient.

By working on a simple component heap using a monadic specification, we are able to avoid many issues associated with the more complicated separation models of programming languages, such as the footprint of objects in a C memory heap (Tuch, 2008) or the inheritance of objects in the Java object heap (Parkinson, 2005; Parkinson and Bierman, 2008).

Granularity The granularity of a separation logic is normally something that is baked into its design. Jensen and Birkedal (2012) developed a separation logic where the granularity can be specified differently for different proofs, albeit with greater complexity in the proofs themselves. We avoid such complexities by carefully designing our separation logic to have the required level of granularity from the start.

2.3 The take-grant protection model

The take-grant protection model (Jones et al., 1976; Lipton and Snyder, 1977; Snyder, 1977; Bishop and Snyder, 1979) is a family of formal models of a protection system where the possible future distribution of authority may be analysed. The key result of the take-grant models, is that security questions such as "can the entity x ever gain read access to the entity y?" is decidable. The take-grant model is in contrast to the HRU model of Harrison et al. (1976), where they showed that the above security question is in fact undecidable for an arbitrary system.

There have been a number of extensions and formalisations of the take-grant model. Our work builds on some of these, and extends them with two new contributions—the addition of shared capability storage as a first class citizen, and an explicit create right to be able to create an entity using a particular identifier. The take-grant protection model that we present in this work is designed as a conceptual, high-level model for the seL4 microkernel, showing authority confinement and simple information flow results.

We explain the take-grant protection model and our contributions to these models in detail in Chapter 4, but give a brief overview of the existing work and how we compare here.

Traditional models Traditional take-grant protection models analysed the access rights that an entity could *explicitly* gain access to (Jones et al., 1976), and the information an entity could *implicitly* gain access to, without necessarily acquiring the direct authority to do so (Bishop and Snyder, 1979). These models distinguished between active and passive entities (that is, subjects and objects), which made the security models more nuanced than the model that we present here.

These models do not examine the concept of entity destruction or the finite nature of memory. Instead, entities are considered to always exist and any entity can create an unlimited number of other entities. By avoiding reasoning about entity destruction and the finite nature of entity identifiers, these models avoid any potential issues of identifier reuse. By allowing entity destruction, we allow the possibility of a new entity being created with the same identifier as a deleted object. This reuse of entity identifiers can greatly complicate the security of a model. We cover these issues of identifier reuse in detail in Section 4.2.

Our work adds the concept of entity destruction and identifier reuse, and does so in a way that models the nature of real-world systems such as seL4 more closely and still allowing the same reasoning about the explicit and implicit access entities can gain.

Diminish-take and SW access models The diminish-take access model (Shapiro, 1999) adds new "diminished" access rights that filter all capabilities access through these diminished capabilities. These additions do not change the decidability of the model, but allow sharing of capabilities between entities to be done in controlled manner. Shapiro also introduced *SW* (Shapiro-Weber) as a formal model of EROS, with a similar authority model to the diminished-take model.

In these models, Shapiro is the first to add entity destruction (present in the HRU formalisation of access control (Harrison et al., 1976)) to the take-grant protection model. He does not allow identifier reuse, that is, when an entity is destroyed, the identifier of the destroyed object cannot be reused by another object. In our work, we allow the identifiers of deleted objects to be reused, which more closely models the behaviour of systems such as seL4 and EROS.

The SW model does not distinguish between active and passive entities (that is, subjects and objects). This is a pattern that we follow in our work.

Elkaduwe et al's security model for seL4 Elkaduwe et al. (2008) where the first to formalise a take-grant protection model in a theorem prover. Their aim was to produce a high-level security model for the seL4 microkernel. Like Shapiro (1999), Elkaduwe et al. did not distinguish between active and passive entities.

Elkaduwe et al. (2008) restricted the creation of entities and required that entities are only able to create other entities if they possess a capability with Create access rights. Their model though still allows an unbounded number of entities to be created, which is unlike the seL4 kernel that it seeks to model. Entity destruction and entity identifier reuse were not considered in this model, which simplified the model considerably.

Our formalisation of the take-grant protection model is based directly on this work. We extend their work by the addition of non-determinism, restricting

object creation to more closely model the seL4 microkernel and the finite nature of memory, and added the sharing of capabilities as a first class citizen. These contributions are explained in detail in Chapter 4.

Chapter Summary

- Existing work on the *initialisation of operating systems* has either been informal, or at best modelled at a very high-level of abstraction with no connection to the underlying operating system code. Our work aims to bridge this gap, developing a formal algorithm for system initialisation that is proven correct by formally connecting it to the existing seL4 proofs.
- *Separation logic* is a technique that has been used to prove properties about a wide variety codebases and models. Rather than developing another separation logic from scratch, we utilise an existing abstract separation logic developed in conjunction with this thesis (Klein et al., 2012). The separation logic we develop uses a variation of the Burstall-Bornat model for object heaps, using a *single* heap of object components. This allows us to elegantly construct a separation logic with a fine level of granularity.
- The *take-grant protection model* is a formal model of a protection system where security questions such as "can the entity x ever gain read access to the entity y?" is decidable. These models traditionally have not reasoned about the finite nature of computer systems, and so have avoided many of the implications that come from having to reuse memory. Our work extends the literature by taking such reuse into account.

Chapter 3

Background

The goal of this thesis is to take a declarative specification of the desired state of a seL4 system, using the capability distribution language capDL, and to verifiably start the system in conformance with this given specification. In this chapter, we introduce the work on which this thesis is based. In particular, we describe:

- **Isabelle/HOL:** In Section 3.1 we introduce formal verification, and the theorem prover Isabelle/HOL, and the notation that we use in the rest of this document.
- **Separation logic:** In Section 3.3 we give a brief introduction to separation logic. We explain separation logic in greater detail in Chapter 6.
- **seL4:** In Section 3.4 we introduce the seL4 microkernel, the capDL language, the capDL kernel model, and the body of proofs that exist for seL4. We introduce the authority confinement and integrity proofs of Sewell et al. (2011) and the information flow proofs of Murray et al. (2013), and show how they have been linked to capDL by Boyton et al. (2013).

These give us the basis for our work, developing a formal algorithm of system initialisation, and showing how we can reason about the correctness of this algorithm, using the existing seL4 proofs. Parts of this chapter have previously appeared in Boyton et al. (2013) and Andronick et al. (2012).

3.1 Formal verification

In this discussion, it is important to work out how it is that we can "trust" our computer systems to behave as intended. Trust in a computer program can come about in a number of ways. We can make sure that the coder is very good, the coder follows certain coding practices, or we could ensure the code is checked by someone else. Alternatively, we could subject our code to a multitude of tests—as

many as we can think of—hoping to uncover any undesired behaviours. Sadly, no matter how many eyes check over a large body of code, nor how many tests we run, neither seem to work—we keep finding bugs in even the simplest of code, such as the bug Google employees found in a binary search algorithm in the Java API, which had lain dormant for nine or so years (Norvig, 2006).

When we want to have greater confidence in the correctness of our software, something more is needed. Pen-and-paper proofs allow confidence to be developed for an algorithm, but require a level of abstraction in which the finer details are often ignored, such as the precise encoding of numbers. Such abstractions can miss bugs, such as the arithmetic overflow present in the above binary search example.

To avoid these issues, various machine aids can be employed. Model checking is a technique where a computer program automatically checks certain properties of a program or specification, such as deadlock freedom or termination. While work is continuing on allowing model checking to scale to larger codebases, it is generally limited to proving simple properties about small software systems.

In this thesis we use formal verification—developing human-written, machine-checked proofs, in the interactive theorem prover Isabelle/HOL. We develop a formal algorithm of what it means to correctly initialise a system, and show the correctness of this algorithm with respect to a formalised API for seL4, which has been shown to be a correct abstraction of the seL4 code. By joining our proofs to a fully-realistic seL4 API, we can have great confidence that we have not abstracted away important implementation details in our algorithm.

The formal algorithm we develop for this thesis, the formal specification of the seL4 API, and all associated theorems in this work have been hand written and machine checked by the proof assistant Isabelle/HOL (Nipkow et al., 2002). Isabelle is a LCF-style (Gordon et al., 1979; Gordon, 2000) proof assistant commonly used for verification. A LCF (or *Logic for Computable Functions*) theorem prover uses a minimal proof kernel that must be trusted and then all other functionality goes through this proof kernel. This approach is not dissimilar from the trusted kernel approach of the seL4 microkernel that we use in this work. This allows us to have a small, trustworthy base, and yet extend the functionality and not compromise the trustworthiness of the theorem prover.

Isabelle/HOL is an instantiation of Higher-Order Logic (or HOL) in Isabelle. Higher-Order Logic is an extension of first-order logic which adds types and quantification over functions. This gives us a proving environment with a rich variety of libraries and mathematic results. We examine the particular syntax of Isabelle/HOL in Section 3.2.

3.2 Syntax and notation of Isabelle/HOL

The Isabelle/HOL notation used in this thesis largely conforms to everyday mathematical notation, lambda calculus, and the conventions of functional programming languages such as Haskell or Standard ML. The main concepts used in the specification are functions, type declarations and non-recursive function definitions.

Theorems Theorems in this thesis are denoted using the following inference notation.

$$\frac{P \longrightarrow Q \qquad P}{Q}$$

◀ **Theorem 3.1**
Modus ponens

This theorem states that, if $P \longrightarrow Q$ and P are both true, then Q is true.

Types and Functions Terms and functions in Isabelle all have a type. We express the fact that a term x has type $'t$ by the notation $x :: 't$. Isabelle supports both concrete types, such as natural numbers (*nat*), and polymorphic types (which are distinguished from normal types by a quote character ′t).

Functions in Isabelle are total. Function application follows the convention of functional programming. That means $f\ x$ (often written as f(x) in mathematics) is the function f applied to argument x. Multiple arguments are delimited by spaces, that is, $h\ x\ (3 + y)$ is the function h applied to two arguments namely x and $3 + y$. The function h could be of type $h :: 'a \Rightarrow nat \Rightarrow 'a\ list$, meaning that it takes a parameter of type $'a$, another parameter of type *nat*, and returns a value of type $'a\ list$. The composition of two functions is defined as $f \circ g = (\lambda x.\ f\ (g\ x))$.

Functions can be partially applied, that is, not all arguments need be given at the call site. For the function h for instance, the term $h\ x$ would denote a function that expects one further parameter. Functions can be applied to a set of values using `, for example, $f\ `\ \{1, 2, 3\} = \{f\ 1, f\ 2, f\ 3\}$.

Functions can be higher-order, that is, they can again take other functions as arguments. For example, the function h of type $nat \Rightarrow (nat \Rightarrow nat) \Rightarrow nat$ is a function whose first argument is a natural number and whose second argument is a

function from *nat* to *nat*. Predicates are functions that return values of type *bool*.

We can update the value of a function using the notation: $f(a := b)$, which stands for the function that at position a returns the value b, and otherwise returns what f would have returned. Its formal definition is $f(a := b) \equiv \lambda x.$ if $x = a$ then b else $f\ x$.

Sets Sets in Isabelle are typed. For example, the set of natural numbers is of type *nat set*. The notation for sets follows that of standard mathematics: set membership is denoted $x \in A$, the empty set $\emptyset$, the universal set UNIV, and the cardinality of a finite set $|A|$. The term $\{a \mid \exists b.\ a * b = 42\}$ denotes the set of factors of 42.

Lists Lists in Isabelle are denoted $[1, 2, 3, \ldots]$. List membership is denoted $x \in xs$, the empty list $[]$, and the length of a list $|xs|$. We append an element to the front of a list using the syntax $x{\cdot}xs$, and concatenate two lists using the syntax xs @ ys. The list of numbers from a to b inclusive is denoted $[a..b]$. As is common in mathematics, to improve readability, we implicitly convert between lists and sets when displaying our formal definitions and theorems in this thesis.

Datatypes and the option type New type constructors can be defined using the **datatype** command. The option type, for instance, is defined as a datatype and often used to add a special element to an existing type to indicate failure or undefinedness. Its definition is

datatype $'a$ *option* = None | Some $'a$

The option type can be used to model partial functions (sometimes called partial mappings). The function f of type $'a \Rightarrow 'b$ *option* takes a value of type $'a$ and optionally returns a value of type $'b$. A lookup function could for instance return None for lookup failure and Some r to indicate a result r. For partial functions we introduce the constant map_empty for the empty map, the notation $f(a \mapsto b)$ as shorthand for f (a := Some b), and the notation $[a \mapsto b]$ for the partial function that maps only the value a to the value b.

Words Isabelle has number of word types. In this work we use *word32* and *word8* which represent unsigned 32 bit and unsigned 8 bit

words respectively. In the notation of this thesis we implicitly convert from a word type to a natural number when necessary.

Pairs The type of pairs (or 2-tuples) is denoted by $'a \times 'b$. Relations can be modelled using the type $('a \times 'a)\ set$, that is, as a set of pairs of the same type. The reflexive, transitive closure of a relation r is denoted r^*.

Records A record is a tuple with named fields. The definition

record point =
x :: nat
y :: nat

introduces a new type point that contains two fields x, and y, both of type *nat*. If p is of type point, the term $x\ p$ is the x-field of p and $y\ p$ the y-field of p, that is, record fields names can be used as accessor functions. The term $(\!|\ \mathrm{x} = 3, \mathrm{y} = 7\ |\!)$ constructs a new record and stands for a point with x-field 3 and y-field 7. To update an existing record p, we write $\mathrm{p}\ (\!|\mathrm{x} := 4\ |\!)$. This stands for the record that has the y-field of p and the x-field 4.

Let binding To simplify definitions, we introduce the let notation, for example,

$\mathsf{let}\ y = x * x; b = y + 1\ \mathsf{in}\ h\ a\ b$

is shorthand for

$h\ (x * x)\ (x * x + 1)$

Common functions

We utilise a number of functions in this thesis, many of which are common in functional languages such as Haskell or Standard ML. We list them below.

$xs_{[n]}$	The nth element in a list xs.
$[x \leftarrow xs.\ P\ x]$	The list xs filtered to only contain the elements for which the predicate $P\ x$ holds.
take $n\ xs$	The first n elements of a list xs (or xs if $n < \lvert xs \rvert$).
drop $n\ xs$	The list xs with the first n elements removed (or $[]$ if $n < \lvert xs \rvert$).
map $f\ xs$	The list where each element has the function f applied to it.
zip $xs\ ys$	Makes a list of corresponding pairs from the two lists. If one list is shorter, then the excess elements from the other list are discarded.

foldl f a xs	Reduces a list xs, from left-to-right using a binary operator f and a starting value a.
foldr f xs a	Reduces a list xs, from right-to-left using a binary operator f and a starting value a.
map_of ps	Turns an association list into a partial map, mapping the first element of a pair to the second element of the pair.
$\bigwedge_{x \in xs} P\ x$	Applies a predicate P to a list of values xs and returns true if the predicate $P\ x$ holds for *all* the elements x in the list xs.

State Monad and Do-Syntax

Since the system initialiser in this thesis is inherently state-based, many functions in the specification modify state in some way. Isabelle/HOL provides a convenient specification mechanism for this that is also known from functional languages such as Haskell, the so-called *state monad* (Cock et al., 2008). A state monad is merely a function from a state $'s$ to a pair of new state and return value $'s \times 'r$. This can be thought of as modelling a usual C function that has a side effect (produces a new state) and returns a value. The syntax for chaining such functions together is the following:

```
do
    a ← f;
    b ← g a;
    h a b
od
```

In the example, first function f is called, returning value a, and potentially changing the underlying state. In this new state $g\ a$ executes, producing result b and potentially using f's result a. The whole do-block returns whatever $h\ a\ b$ returns, and produces the state after execution of h. This means the state is threaded through the functions implicitly, while the result values are passed explicitly. The type of each of the single functions is still $'s \Rightarrow 's \times 'r$, potentially with additional arguments if they make use of previous result values.

The idea is to make the syntax reminiscent of an imperative programming style while staying in a strictly functional setting where the state and its type could be made explicit if convenient. It also allows the type checker to enforce that certain functions only modify certain parts of the state.

The library this specification is based on extends such state monads with non-determinism and a mechanism to raise and check assertions. Non-determinism basically means that functions do not return only one state and result, but a set

of possible new states and results. The type of such functions is abbreviated as $('s, 'v)$ *nondet_monad* where $'s$ is the type of the state and $'v$ the type of the result value. If the function does not return a result value, the type *unit* is used for $'v$. In this thesis, as most functions do not return a value, we omit the return value from the presentation of our theorems for brevity.

We introduce here a number of basic monad functions:

gets	expects a function $'s \Rightarrow 'v$ as argument and returns the result of applying this function to the state. It does not modify the state. It is usually used to extract one field from the state when the state is modelled as a record. [1]
assert P	fails if the predicate P is false, and returns nothing if it is true.
assert_opt v	fails if the value v is None, and returns x if it is Some x.
return	returns its argument and does not modify the state. It is used to perform computations that are state-independent.
f \$ x	applies function f to argument x. This alternative syntax for function application makes it possible to write fewer parentheses. For example, f \$ x + v is equivalent to f (x+v).
mapM f xs	executes a monadic function f on each element of the list xs in order, discarding the return result.
whileLoop c B i	applies the body of a loop B, as long as the condition c is true, given an initial loop iterator value i. This combinator is taken from Greenaway et al. (2014).
inc_when P x	when P is true, returns $x + 1$, otherwise returns x.
update_when P f a b	when P is true, returns $f(a \mapsto b)$, otherwise returns f.

As stated before, the full definitions, theorems, and proofs contained in this thesis are all published under an open-source licence (Trustworthy Systems Team, 2014).

[1] As the system initialiser specification itself has no state, only an embedding of the kernel state which it does not read (as will be explained in Section 7.4), it never uses gets. The capDL kernel model does, however, get (and modify) the kernel state and so does use gets (and the corresponding monadic operation modify).

3.3 Separation logic

To reason about the correctness of our system initialiser algorithm, we need to reason that each step of our algorithm creates and/or configures a specified object, and that this step does not undo any of the work done by previous steps. To do so, we use separation logic, an extension of Hoare logic (Hoare, 1969) first proposed by Reynolds (2002).

Separation logic was first introduced for local reasoning about imperative programs that access and modify computer memory, and is commonly used to reason about concurrent programs. We use separation logic in this thesis to reason about our formal model of system initialisation and how it transforms the objects in the seL4 kernel state. We have found that it is an elegant fit for this style of reasoning. The local reasoning provided by separation logic allows us to easily specify exactly which objects are modified by the system initialiser and which objects remain unchanged. We cover this in more detail in Chapter 6.

3.4 The seL4 microkernel

In this section, we introduce the capability-based microkernel seL4 on which we base this work. We introduce seL4, the existing formal verification work done on seL4, and capDL—a high-level formalisation of the state and semantics of the seL4 kernel.

seL4 overview

The seL4 microkernel is a general-purpose operating system (OS) kernel designed as a secure and reliable foundation for a wide variety of applications. An OS kernel is the only software running in the *privileged* mode of the processor. The seL4 microkernel is formally verified for full functional correctness to both the C code and the binary level on the ARM platform (Klein et al., 2009; Sewell et al., 2013). This means that there exists a machine-checked proof that both the C code and binary of seL4 are a correct refinement of its functional, abstract specification, as explained in Section 3.4.

As a microkernel, seL4 provides a minimal number of OS services: threads, inter-process communication, virtual memory, and capability-based access control. Throughout this thesis we will use the seL4 system illustrated in Figure 3.1 as a motivating example for this work. We also use this example in this section to explain the OS services that seL4 provides. This system contains two threads, a sender `A` and a receiver `B`, communicating via a shared *endpoint* `EP`, and a number of other objects which we will explain below.

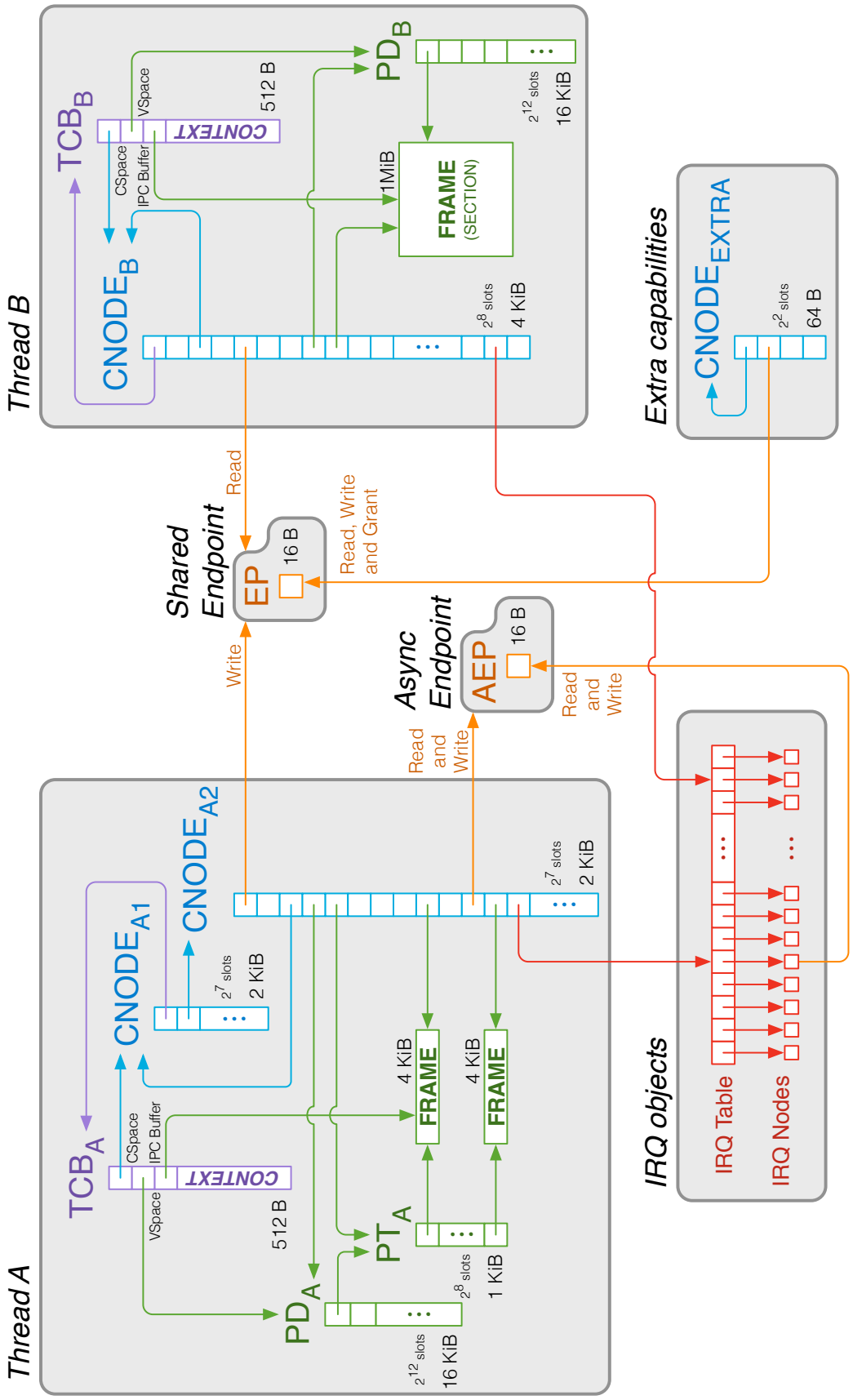


Figure 3.1: A seL4-based system with two threads that can communicate via an *endpoint*, with all capabilities shown (where capability slots that are not shown are empty). All capabilities are assumed to have full access rights unless explicitly shown otherwise.

Each thread is represented by its *thread control block* (TCB), which stores its context, virtual address space (VSpace), capability space (CSpace), and IPC buffer[2]. A VSpace defines the memory accessible to the thread; it is represented by a set of frames, generally organised in a hierarchical, architecture-dependent structure of page tables and page directories.[3] Each virtual address space in seL4 is assigned a unique address space identifier (ASID). CSpaces are kernel managed storage for *capabilities*. A capability is an unforgeable token that confers authority to an object. They are stored in a graph of capability nodes (CNodes) and are explained in detail in Chapter 4. In seL4, when a thread invokes an operation on an object such as sending a message to an endpoint using the seL4 system call `seL4_Send()`, it needs to provide an index into its CSpace that indexes a capability that possess sufficient authority to that object. For instance, for sender `A` to send a message to sender `B`, sender `A` needs a *write* capability to a shared endpoint, while receiver `B` needs a *read* capability to the same endpoint. Similarly, for a thread to be able to configure itself, its capability address space, or its virtual address space, it requires a capability with sufficient permissions to its thread control block, its CNodes, or its virtual memory objects[4] respectively. In our example, both threads have these access rights.

seL4 supports three communication primitives: *synchronous endpoints* (often referred to simply as *endpoints*), *asynchronous endpoints*, and shared memory. A sender may send a message over an endpoint, and a receiver can wait on an endpoint for a message. Sending a message over a synchronous endpoint is a blocking operation where the sender is blocked until the receiver receives the message. Sending over an asynchronous endpoint is a non-blocking operation.

Capabilities in seL4 can be transferred in two ways. Firstly, a copy can be sent over a synchronous endpoint (but not an asynchronous endpoint) in a traditional take-grant style approach (see Chapter 4) if the threads possess sufficient permissions.[5] Secondly, a capability can be transferred between threads directly if the two threads share a common CNode. Both of these notions of capability transfer are formalised in Chapter 4.

Asynchronous endpoints are also used for the delivery of *hardware interrupt requests* (IRQs). A thread with sufficient access privileges to a specific IRQ line (namely an *IRQ Handler Capability* to that IRQ line) can bind an asynchronous

[2]The seL4 instruction to be executed by a thread is encoded in the registers of a thread and the IPC buffer for the thread.

[3]In this thesis we reason about ARM-based systems, although the concepts work on both ARM and x86 platforms. Sizes shown are for ARMv6. Note that on ARM-based systems, a 1MiB frame is called a *section*.

[4]Virtual memory objects consist of page directories, page tables, and frames.

[5]In Figure 3.1, neither thread is able to transfer capabilities this way as neither possess a capability to an endpoint with *grant* permissions.

endpoint to that IRQ line. The thread is then able to *wait* on this asynchronous endpoint until an interrupt is received. Internally, seL4 stores an array of IRQ Nodes, one node per interrupt line, and when an asynchronous endpoint is bound to an IRQ line, a capability to that endpoint is stored in the corresponding IRQ Node. seL4 stores the correspondence from IRQ lines to IRQ Nodes in an IRQ table. These concepts are illustrated in Figure 3.1.

An index into a CSpace is referred to as a *capability pointer*, and is decoded to point to a capability slot in a CNode in a similar way as to how a virtual address is decoded to point to a page in a guarded page table structure (Elkaduwe et al., 2006). As the creation and traversing of capability spaces is an important part of this thesis, we briefly explain how the decoding of a capability pointer to a capability slot is done. A full understanding of the decoding algorithm is not required, but a short example of capability decoding should hopefully assist the reader in understanding the complexities of seL4's capability spaces. To decode the capability pointer 0x00200000, using a depth[6] of 32 bits, in the capability space illustrated in Figure 3.2, we first strip the first 4 bits of the capability pointer (0x0) as a guard[7] (checking that the guard does indeed match the first 4 bits), leaving 0x0200000. The next 8 bits (0x02) are an index into the top level CNode, CNode_1, where we find an Endpoint Cap at the slot with index 0x02. Since we have reached a capability that is not a CNode capability, we terminate the decoding, and thus the capability pointer 0x00200000 points to this Endpoint Cap.

Similarly, to decode the capability pointer 0x0FF20000, with a depth of 32 bits, we strip the first 4 bits (0x0) off as a guard as before, the next 8 bits (0xFF) index the top level CNode, CNode_1, and point to the CNode capability located in slot 0xFF. Since we have not translated 32 bits yet and have another CNode capability that we can follow, we continue to decode the rest of the capability pointer, namely 0x200000. The CNode Cap in slot 0xFF of CNode_1 has an empty guard and points to CNode_3, so we use the next 4 bits (0x2) as an index to find the TCB Cap in the CNode_3. Since this capability is not a CNode, we terminate the decoding here as before, and the capability pointer 0x0FF20000 points to this TCB Cap.

Capability spaces can be cyclic, allowing the capability pointer 0x0FFE1FF2 to also point to the same TCB capability as the capability pointer 0x0FF20000 does (an exercise that we will leave to the reader). As we said earlier, a full understanding of seL4's liberal capability storage semantics is not required to understand this thesis, but it suffices to know that they are complicated, and that any process that wants to distribute these capabilities must be careful to do so correctly. Whereas

[6] To translate a capability pointer, we always need to include a depth. The depth states how many bits of the capability pointer should be decoded. The depth is specified by a user when specifying a capability pointer.

[7] A CNode capability has a guard, which is a (potentially empty) number of bits.

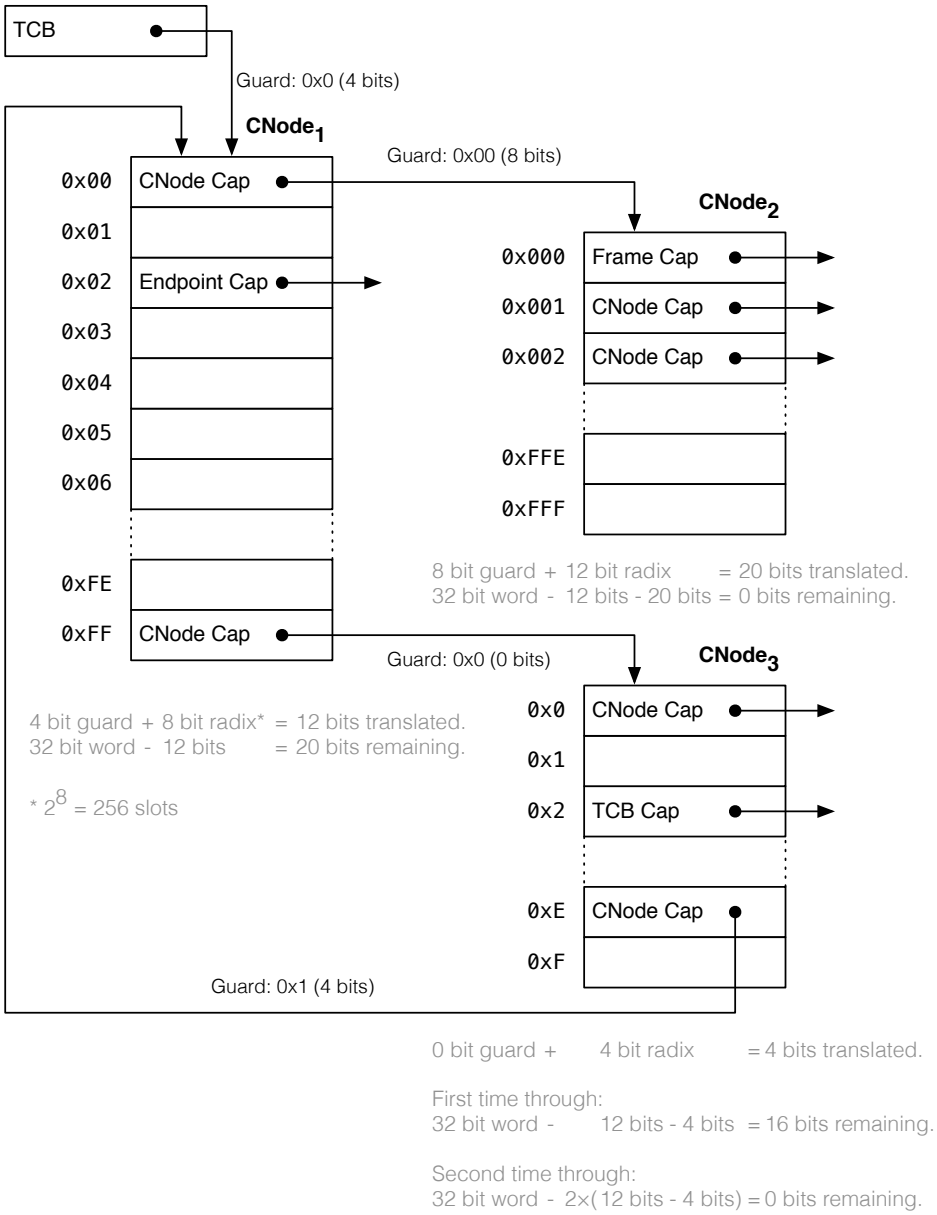


Figure 3.2: An example capability space.

most users of seL4 will most likely never create such an intricate capability space, the initialiser that we develop for this thesis should be able to support the creation of them.

To be able to create a system such as the one described in Figure 3.1, seL4 allows the allocation of kernel objects by the retyping of *untyped memory*, an abstraction of a region of physical memory. Possessing a capability to a region of untyped memory confers the authority to allocate kernel objects, such as a CNode or an Endpoint, in this region. At boot time, seL4 first pre-allocates memory for itself and then gives the remainder to the initial user task in the form of capabilities to untyped memory. To create the objects described in Figure 3.1, the initial user task would *retype* this untyped memory to create the required objects by using the `seL4_Untyped_Retype` system call.

This initial user task (sometimes called the *root task*) is the initialiser we are targeting in this thesis. Its aim is first to use these untyped capabilities for creating the required objects, such as TCB_A, TCB_B, and $\mathsf{CNode}_{\mathsf{A}_1}$ from Figure 3.1, and then to initialise them appropriately, for example, to set TCB_A's CSpace to $\mathsf{CNode}_{\mathsf{A}_1}$. This includes setting up communication channels, for example, storing the *write* capability to `EP` in TCB_A's CSpace.

When seL4 allocates a new object, it creates a new capability with full access rights to the newly created object. Further copies of the original capability may be *derived* from the original capability (or from other derived copies), either through explicit copying, or sending a copy of a capability through an Endpoint. seL4 tracks this derivation of capabilities in a data structure called the capability derivation tree (CDT) (Elkaduwe et al., 2007). The derived capabilities are stored as *children* of the capability from which they are derived (which would be the *parent*). Possessing an original capability confers more authority than a derived capability—a user can revoke all copies of an original capability (by calling `seL4_CNode_Revoke`), which deletes all derived capabilities. For this reason, it is important to track the difference between original and derived capabilities. As mentioned earlier in this section, these copies of a capability can be made by sending a copy of a capability through an Endpoint using the `seL4_Send` system call, or by copying the capability into another capability slot using the `seL4_CNode_Copy` or `seL4_CNode_Mint` system calls (where minting a capability allows the copy to have less access rights than the original).

In our example, we wish to have one thread with *Read* access to the shared endpoint `EP`, and the other with *Write* access. The original capability to this endpoint (which possesses full access rights to the endpoint, namely, *Read*, *Write*, and *Grant*) needs to be stored somewhere. In our example we have placed this

"extra" capability in another CNode object $\mathsf{CNode}_{\mathsf{Extra}}$. This extra CNode also needs a capability pointing to it, and so we store this capability in the CNode itself.

This creation of another CNode is not only convenient for initialisation (as it allows us to avoid reasoning about capability deletion), it is necessary for the security properties of seL4 shown by Sewell et al. (2011) (which are introduced in detail in Section 3.4). Sewell et al. (2011) showed that an object with a *parent* capability possesses rights over another object that possesses a *child* capability derived from the parent capability (as encoded by the CDT). For example, seL4 allows the holder of a parent capability to revoke derived capabilities.

If two capabilities are derived from the original capability, like in our example where the two threads both have capabilities to the Endpoint derived from the original stored in $\mathsf{CNode}_{\mathsf{Extra}}$, and then the original capability is deleted, then one of the derived capabilities becomes the *parent* capability of the other, which could be undesirable. For this reason, we do not delete the original capabilities in initialisation, and instead allow the user of the system to decide where to put these original capabilities. Since this CNode object $\mathsf{CNode}_{\mathsf{Extra}}$ is isolated from the other objects, neither thread is able to use these original capabilities, which removes these security risks. We do not make explicit in Figure 3.1 which capabilities are the original capabilities. We instead make this information explicit in our formalisation of this example using capDL in Section 3.4.

seL4 verification and capDL

As stated in the start of this chapter, the seL4 microkernel is formally verified for full functional correctness to the binary level, as illustrated in Figure 3.3. This functional correctness proof is done through refinement, a formalised proof that the seL4 kernel implements an abstract specification of its behaviour. This refinement proof first joins the abstract specification of seL4 to a formal model of the seL4 source code (using Isabelle/HOL) (Klein et al., 2009), and then to a formalisation of the seL4 binary (using Isabelle/HOL, HOL4 and SMT solvers such as Z3) (Sewell et al., 2013).

On top of this functional correctness proof, Sewell et al. (2011) showed that seL4 maintains authority confinement and integrity. Authority confinement provides an upper bound on how authority may change, similar to the properties of authority confinement we show of the take-grant model in Chapter 4, whilst integrity provides an upper bound on write operations. They showed that, for any initial state conforming to a user-specified access policy, seL4 will, a) ensure that this policy is always obeyed, and b) ensure that the state will always be in conformance with this policy (given a number of assumptions specified in their paper). These access policies state precisely which rights certain object possess over other

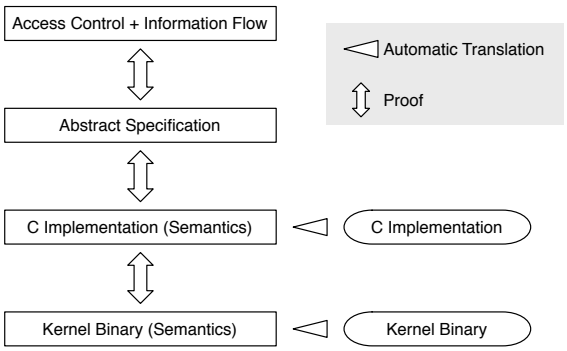


Figure 3.3: The refinement layers in the verification of seL4 functional correctness.

objects. Murray et al. (2013) later extended this work to show noninterference for the seL4 kernel, and showed exactly how to configure seL4 to provide this noninterference. These results, while proven on the abstract specification, hold on the C implementation and the binary via refinement. For a summary of these results, and their implications, we point the reader to Klein et al. (2014). These results both assume, however, that the system is started in conformance with an access control policy, the process of which is the focus of this work.

A system does not imply a single access policy, but instead we can specify an access control policy for a state, and then can show that our system implements the policy. As a concrete example of what one of these access policies can look like, we label the objects of our example state from Figure 3.1 in Figure 3.4. The grey boxes show objects labelled with the same label and they grey arrows between the labelled regions show the access permissions between the regions. A user is then able to show, using the work of Boyton et al. (2013), that this capDL state does implement this access control policy.[8]

This work of Sewell et al. (2011), Murray et al. (2013), and Boyton et al. (2013) then allow us to reason about the access control, integrity and information flow of this system at a high-level of abstraction, and know that these results hold on the kernel implementation.

Despite the capability-oriented design of many microkernels, they generally contain authority relevant to information flow and access control that is not conferred by capabilities.[9] For example, in our example specification in Figure 3.1, in the seL4 implementation the page table PT_A, is mapped into the page directory PD_A not with a capability but with a pointer. Additionally, when a thread sends a

[8]The labels and authority of this example have not been proven, but are included as a guide of what is possible.

[9]Which is why the work of Sewell et al. (2011) and Murray et al. (2013) examined more than the capabilities of the system, but also other non-capability data, such as page directory and page table mappings.

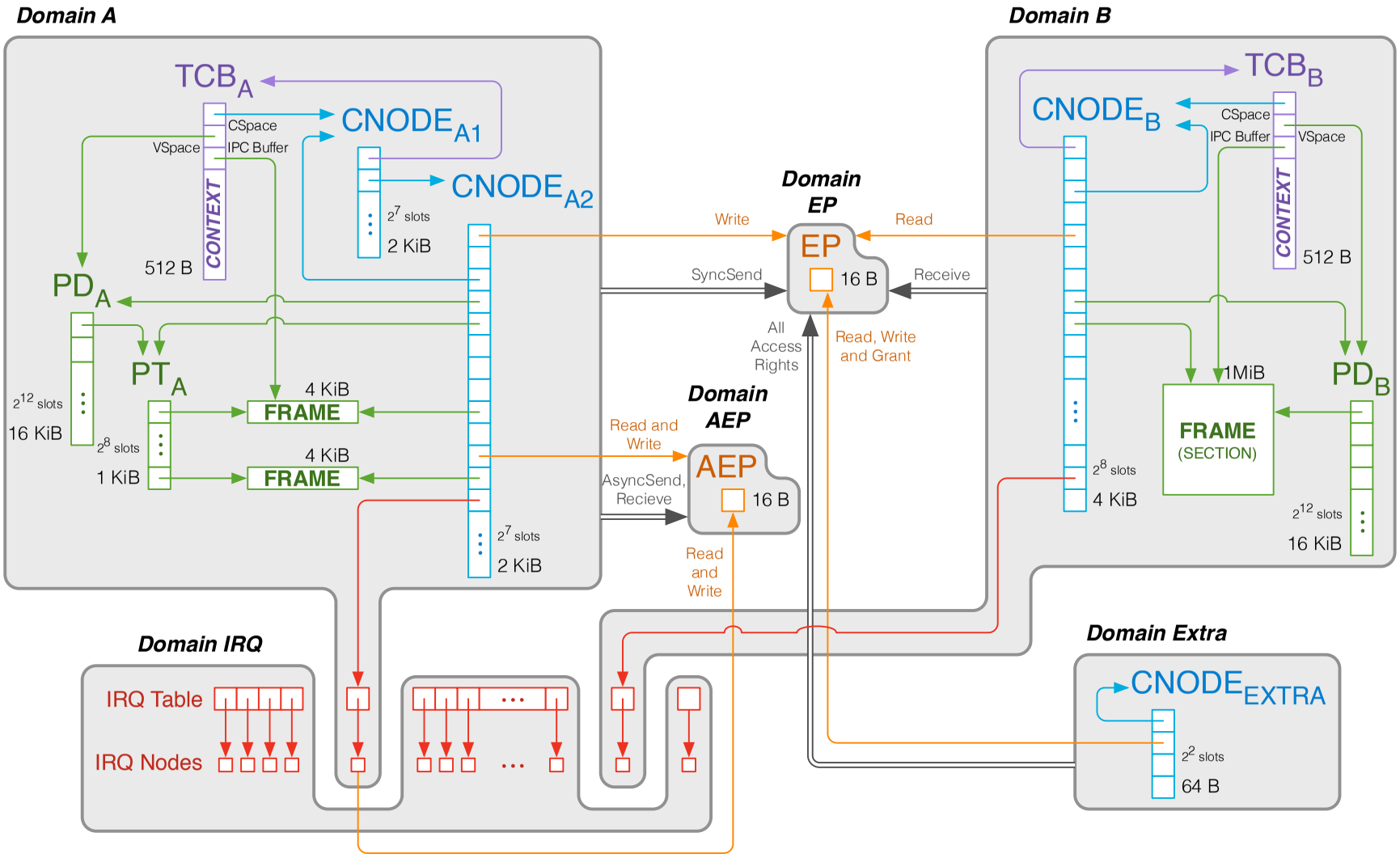


Figure 3.4: The example state Figure 3.1, labelled with access control domains.

message through a synchronous endpoint, the authority to do so is checked upon the start of sending the message. If the thread is blocked after starting to send this message (as is the case when the receiver is not waiting for it), then the thread sending has an implicit *right* to conclude sending this message, even if the explicit right is removed whilst the thread is blocked. These implicit rights present in seL4 are detailed in Sewell et al. (2011).

Having all the protection state described by capabilities would enable reasoning about the access control and security of a system through capability distributions alone. It would also allow components and connections of user-level systems on top of seL4 to be described by their capability distribution alone. Such a description is basically a graph with objects as nodes and capabilities as edges. To reason about such specific graphs, the capability distribution language capDL (Kuz et al., 2010) was developed. The capDL language unifies all information relevant to information flow and access control as explicit capabilities.

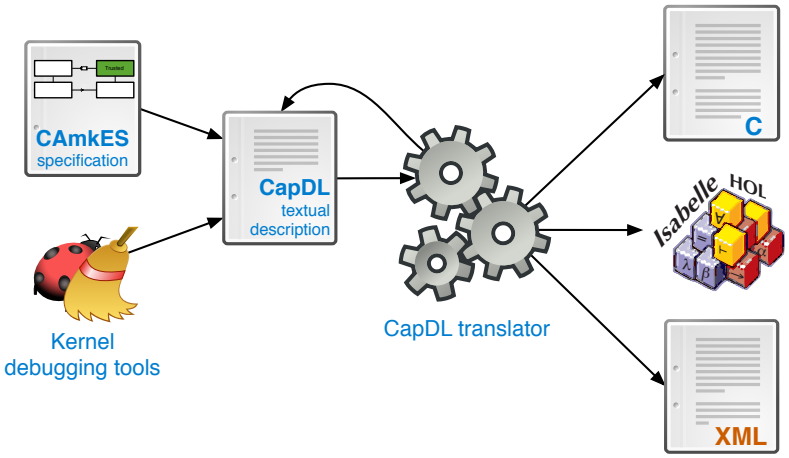


Figure 3.5: Some of the suite of tools for producing a capDL specification

Kuz et al. (2010) developed a formal semantics for the system state in Isabelle/HOL, and a suite of tools for describing these capDL specifications, as illustrated in Figure 3.5. To initialise a system such as the one shown in Figure 3.1, a user can specify the configuration in a textual description, like the one shown in Figure 3.6, and import the capDL description into Isabelle/HOL, where they can analyse the corresponding access control policy in the theorem prover Isabelle/HOL. These tools can also generate a C data structure that is used by the system initialiser to start a system in conformance with the capDL specification, as described in Chapter 5, and export the specification in an XML representation for use by other tools. These capDL textual descriptions can be written by hand, or they can be generated

```
arch arm11

objects {
    tcb_a = tcb (addr: 0x00000000,
                 ip: 0x00000F10,
                 sp: 0x00000100,
                prio: 125)

    tcb_b = tcb (addr: 0x00001000,
                 ip: 0x00002E00,
                 sp: 0x00100000,
                prio: 125)

    cnode_a1 = cnode (16 bits)
    cnode_a2 = cnode (16 bits)

    cnode_b  = cnode (12 bits)

    cnode_extra = cnode (2 bits)

    pd_a = pd
    pt_a = pt
    pd_b = pd

    frame_a1 = frame (4k)
    frame_a2 = frame (4k)
    frame_b  = frame (1M)

    ep_shared = ep
    aep_irq = aep

    irq_node_0x04 = irq
    irq_node_0xFE = irq
}


caps {
    tcb_a {
      cspace: cnode_a1 (guard: 0, guard_size: 28)
      vspace: pd_a
      ipc_buffer_slot: frame_a1 (RW)
    }

    tcb_b {
      cspace: cnode_b (guard: 0, guard_size: 28)}
      vspace: pd_b
      ipc_buffer_slot: frame_b (RW)
    }

    cnode_a1 {
      0x0: tcb_a
      0x1: cnode_a2 (guard: 0, guard_size: 28)
    }

    cnode_a2 {
      0x0: ep_shared (W)
      0x2: cnode_a1 (guard: 0, guard_size: 28)
      0x3: pd_a
      0x4: pt_a
      0x8: frame_a1 (RWG)
      0xA: aep_irq (RW)
      0xB: frame_a2 (RWG)
      0xC: irq_node_0x04
    }

    cnode_b {
      0x00: tcb_b
      0x02: cnode_b (guard: 0, guard_size: 28)
      0x04: ep_shared (R)
      0x07: pd_b
      0x08: frame_b (RWG)
      0xFE: irq_node_0xFE
    }

    cnode_extra {
      0x1: cnode_extra (guard: 0, guard_size: 28)
      0x3: ep_shared (RWG)
    }

    pd_a {
      0x0: pt_a
    }

    pt_a {
      0x0: frame_a1 (RWG)
      0x4: frame_a2 (RWG)
    }

    pd_b {
      0x0: frame_b (RWG)
    }

    irq_node_0x04 {
      0xA: aep_irq (RW)
    }

    irq_node_0xFE {
    }


cdt {
    (cnode_extra,0x0) {(cnode_a2, 0x0); (cnode_b, 0x3)}
}

irq maps {
    0x04: irq_node_0x04
    0xFE: irq_node_0xFE
}
```

Figure 3.6: CapDL textual description of the state of Figure 3.1.

automatically from high-level architecture specifications such as CAmkES (Kuz et al., 2007), or objects from a running kernel using debugging tools.

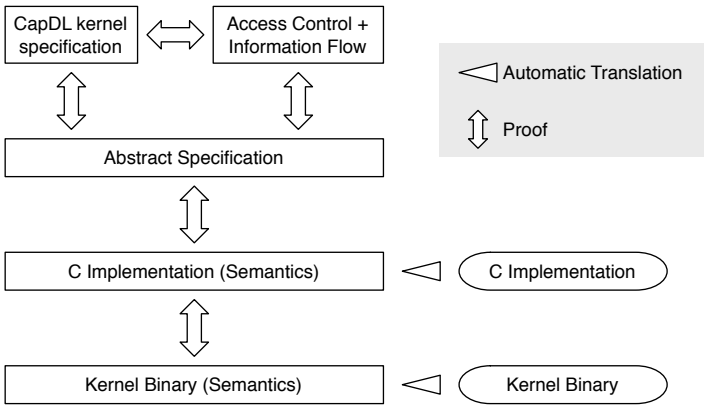


Figure 3.7: CapDL model in the seL4 refinement chain (where arrows denote formal proof).

Boyton et al. (2013) showed that capDL did in fact meet its design aim by showing that we can describe complete access control system configurations (as formalised by Sewell et al. (2011)) by capability distributions alone (by using capDL). They did this by showing that capDL descriptions can be mapped to a corresponding access control policy, as shown in Figure 3.7.

This result justifies the use of capDL to describe the state of a system that we wish to start a system in conformance with as we know that it captures the authority of a system (as formalised by Sewell et al. (2011) and Murray et al. (2013)) through its capability distributions, and we know that seL4 will enforce this access control on the resultant system. This means that each of the arrows in Figure 3.1 are capabilities in capDL, rather than a combination of capabilities and pointers in the seL4 source code.

In addition to the language itself, which describes snapshots of system states (which we call *capDL system descriptions*), Boyton et al. (2013) developed kernel semantics for the capDL language that describes the effect of each kernel operation on such states, and formally showed that this *capDL kernel model* is a correct abstraction of existing models of seL4, with a complete refinement chain to the binary level, as shown in Figure 3.7. This ensures that the seL4 kernel implements the formal specification of the capDL kernel model.

The capDL kernel model abstracts the way that system calls are performed, encoding the system call to be performed by a thread as an *intent*, rather than encoding it as a series of bits scattered between the registers of a thread and the IPC buffer.

It is into this framework that this thesis fits. First, we developed a formal model for system initialisation, that takes a capDL system description as input

of the initialiser, and we show that if the initialiser terminates, if does so in an initialised state corresponding to the capDL description given as input. Secondly, we prove this is the case by joining the proof of the system initialiser to a formal specification of the seL4 API (the capDL kernel model) as explained in Section 7.4.

To prove the correctness of the system initialiser algorithm, and the seL4 API specifications, we use a separation logic we developed for this thesis, as explained in Chapter 6.

Chapter Summary

- Formal verification using the theorem prover *Isabelle/HOL* allows us to trust the correctness of our software.
- To reason about the correctness of our formal algorithm we need to reason that each step correctly creates or configures an object, and also that each step does not undo the work done by the previous steps. We find *separation logic* is a convenient way of doing this style of reasoning.
- The *seL4* microkernel can be used to construct systems with guarantees about the authority confinement, integrity, and information flow. If a system is started in conformance with an access control policy, then seL4 has been formally proven to implement this policy.
- The capability distribution language (capDL) can be used to describe the protection state of a system that we wish to initialise, and has been formally linked to the access control policies of the above authority confinement, integrity, and information flow proofs. The system initialiser that we describe in this thesis takes a capDL description as input, and starts a system in conformance with the description.

Chapter 4

Capability-Based Access Control

Software systems are becoming increasingly complicated, often built from software components created by various third parties. To be able to have assurance that our software systems are worthy of the trust that we increasingly place on them, we want to make sure that these components are given precisely the access that they require, and no more.

Capabilities, first introduced by Dennis and Van Horn (1966), allow us to name and specify this access control at a fine level of granularity. A capability is an unforgeable, explicit set of access rights to an entity, such as a page in memory or a communication channel between components. Capabilities can be used confer both the authority to communicate and the authority to transfer capabilities.

Analysing the security of an arbitrary system was shown to be undecidable by Harrison et al. (1976), who showed that it was equivalent to solving the halting problem. This result led to the creation of the take-grant protection model where various security questions, such as authority confinement, are decidable.

In the remainder of this chapter, we introduce the traditional take-grant protection model in Section 4.1, and we extend the take-grant protection model with two new additions in Section 4.2. We have formalised this extended take-grant protection model in the theorem prover Isabelle/HOL. We first formalise the state of the model in Section 4.3, and the operations in Section 4.4. We then show a number of properties of the model, namely authority confinement of entities in Section 4.5, the authority confinement of groups of connected entities in Section 4.6, and we extend this to reason about the flow of information between these groups of entities in Section 4.7. Finally we relate this work to other research on the take-grant protection model in Section 4.8. All theorems in this chapter have been formalised in Isabelle/HOL.

4.1 Take-grant protection model

The take-grant protection model (Jones et al., 1976; Lipton and Snyder, 1977; Snyder, 1977; Bishop and Snyder, 1979) is a family of formal models of a protection system where the possible future distribution of authority may be analysed. The take-grant protection model is a graph-based model, where entities are nodes of a graph, and capabilities are the edges of the graph (labelled with their access permissions). An example state with three entities, e_x, e_y, and e_z, is illustrated in Figure 4.1. In this example, the entity e_x has Write access to the entity e_y and Take access to e_z, and e_y has Read access to e_z.

These models have an alphabet of access rights comprised of Take and Grant, and a finite number of *inert* access rights as fits the context such as Read and Write.[1] For example Snyder (1981a) adds an Append right that allows information to be added, but not destroyed.

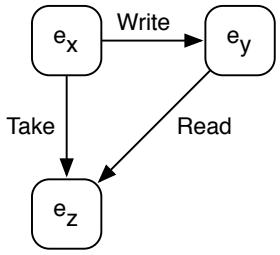


Figure 4.1: An example take-grant system.

Early work by Jones et al. (1976) analysed which access rights an entity is able to *explicitly* gain access to. Bishop and Snyder (1979) later analysed the information an entity can *implicitly* gain access to, without necessarily acquiring the direct authority to do so. The former were retrospectively called *de jure* rights (those "rightfully" acquired) and the latter *de facto* rights (those that exist in fact, whether by right or not). We analyse de jure rights in Section 4.5 and de facto rights in Section 4.7.

The take-grant protection model has four key operations—take, grant, create, and remove. All operations being performed by an entity must be authorised by a capability that entity possesses (except entity creation, which is always authorised in most take-grant protection models[2]). The take-grant protection model seeks only to analyse the distribution and movement of capabilities, and does not

[1] Very early work (Jones et al., 1976; Lipton and Snyder, 1977) used the terms Read and Write for what was subsequently called Take and Grant respectively. This changing of naming occurred very early (Snyder, 1977). We follow the later conventions here.

[2] Restricting entity creation was first done by Elkaduwe et al. (2008) as discussed in Section 4.2.

examine the reading and writing of data (although read and write *capabilities* are generally present). We define these operations as follows:

Take An entity e_x with a capability with a Take access right to another entity e_y can take a copy of one of that entity's capabilities α[3], as illustrated in Figure 4.2.

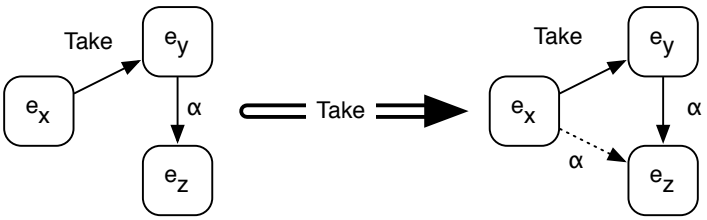


Figure 4.2: The take operation

Grant An entity e_x with a capability with a Grant access right to another entity e_y is able to grant a copy of one of its capabilities α to that entity, as illustrated in Figure 4.3.

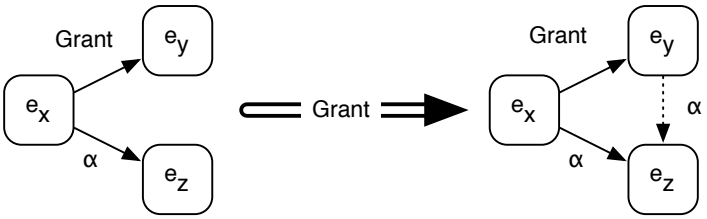


Figure 4.3: The grant operation

Create Any entity e_x can create a new entity e_n, to which it has full access rights, as illustrated in Figure 4.4.

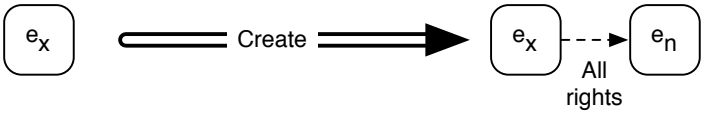


Figure 4.4: The create operation

Remove An entity e_x can remove one of its capabilities α, as illustrated in Figure 4.5.

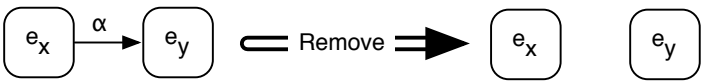


Figure 4.5: The remove operation

[3]We follow the literature and use α to mean any arbitrary capability.

We can use this model to both build systems and analyse what capabilities an entity can acquire.

The key result of the take-grant protection model is that the security question "is it true that p can α y?"[4] is decidable (Jones et al., 1976). A somewhat surprising feature of this model that Jones et al. (1976) showed is that, assuming all entities work together, an entity e_x with a Take right to another entity e_y can transfer an arbitrary capability α to that entity, as illustrated in Figure 4.6.

4.2 Extensions to the take-grant protection model

Create access right

When implementing a capability-based system each entity requires an identifier such as the memory address of the entity.

In a system where entities are created and destroyed, we either have the option of reusing entity identifiers, or not. If we never reuse the identifiers of entities we always have the risk of exhausting the finite supply of identifiers[5]. Alternatively, reusing entity identifiers allows the possibility of impersonation of entities. For example, let us consider the case where we have two entities, the entity with the identifier e_x, and the entity with the identifier e_y[6]. If the former entity has a Read capability to the latter, and these are the only entities and capabilities in the system, then the entity with identifier e_x, should never be able to gain any more access to the entity with identifier e_y, and cannot in the standard take-grant protection models.

Yet, if entity destruction is allowed, the entity with identifier e_x is able to destroy the entity with identifier e_y, and create a new entity with identifier e_y, to which is will have full access rights, as illustrated in Figure 4.7a. This behaviour is certainly undesirable, as the entity with identifier e_x is not actually able to gain full access to the entity with identifier e_y, but instead a new entity with the identifier e_y. This precludes us from making statements of the form "the entity with identifier e_x is never able to access the entity with identifier e_y", which are exactly the sort of statements we wish to make.

This problem is not new, for example, Wu et al. (2013) have the same problem when formalising the Role-Compatibility Model, and so their completeness theorem for taint analysis is restricted to *undeletable* files (which generally would be system files).

[4] Here α can be any access right, such as Read or Write.

[5] In real-world systems, computers are finite.

[6] Specifying "the entity with identifier e_x" is cumbersome, and will normally be called "the entity e_x", but the distinction is important here.

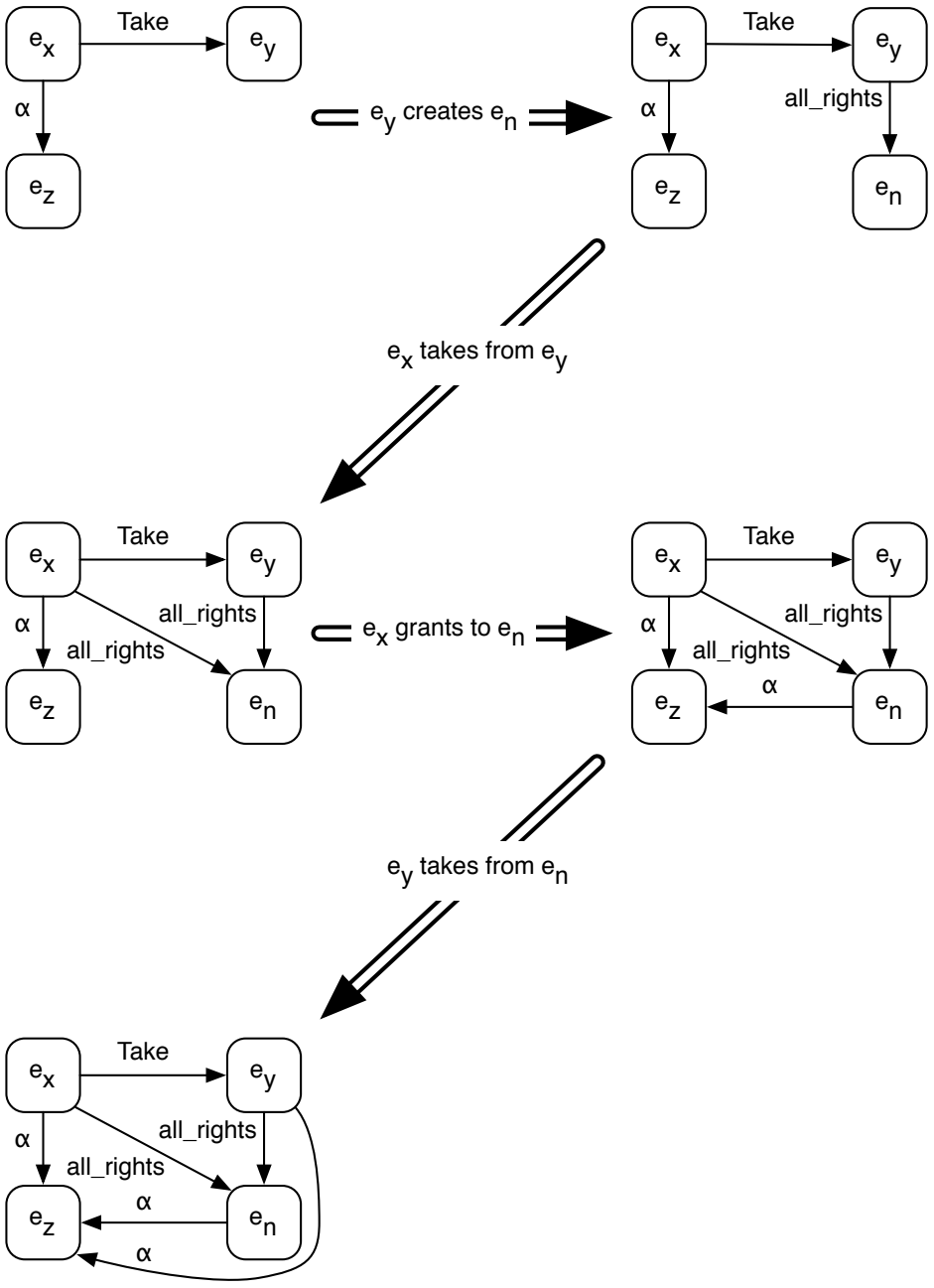
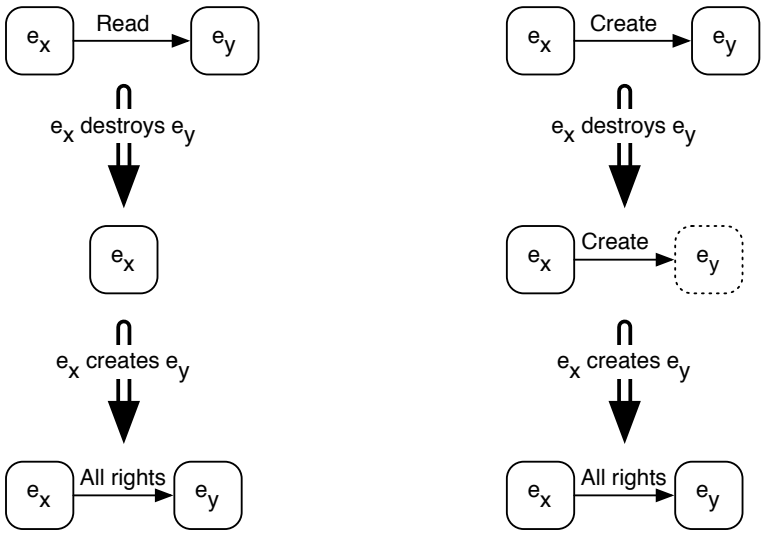


Figure 4.6: A sequence of operations that transfer a capability from e_x to e_y.



(a) Traditional take-grant models (extended with entity destruction)

(b) New take-grant model with explicit Create access rights

Figure 4.7: A example illustrating the dangers of identifier reuse. Entity e_x is able to gain full access to e_y.

It is for this reason, that Shapiro, in his SW model of EROS (Shapiro, 1999)—an extension of the take-grant protection model, added a destroy operation that did not allow identifier reuse.[7] Real world capability-based systems, such as the EROS microkernel (which Shapiro was modelling) and the seL4 microkernel (which we aim to model here) do allow entity destruction *and* entity identifier reuse.

Rather than avoiding this problem, we have extended the standard take-grant model (which did not support entity destruction at all) to allow both entity destruction and entity identifier reuse. We do this by adding a new, non-inert access right Create, which confers the authority to create a new entity with the specified identifier. Adding a Create access right was first done by Elkaduwe et al. (2008) to model some of the restrictions that the seL4 microkernel has on entity creation. These restrictions are present in seL4 to avoid the exhaustion of memory by an entity creating an arbitrary number of other entities (Elkaduwe et al., 2006). The Create access right in Elkaduwe et al. (2008) conferred the authority to create new entities by restricting an entity's ability to create so that they could only do so if they possessed a capability with Create access rights. They did not model the finite nature of memory as possessing a capability with Create access rights allowed such an entity to create an unbounded number of new entities. They also did not allow entity destruction, and thus did not consider the problem of entity identifier reuse.

[7] "The formal evaluation will finesse this issue by assuming that there is a large finite pool of unallocated objects, and that these objects are never reused." (Shapiro, 1999)

Instead we propose that the Create access right confers the authority to create an entity with a specific identifier. This allows us to solve the identifier reuse problem, as well as modelling the behaviour of seL4's Untyped capabilities more closely.[8]

In our model, an entity is only able to create or destroy an entity when it possesses a capability with Create rights to that entity identifier. In this way, the possible entity identifiers within a system are fixed, as actually they are in a computer system (even if this is every possible byte in memory). The destruction and recreation of an entity allows an entity to gain full access to an entity with the same identifier as any that it possesses a capability with Create rights to. Thus we consider possessing Create rights to an identifier to be the same as possessing full access rights to the entity located at that identifier. This is formalised in Section 4.5. We believe that this is the first take-grant protection model that adequately addresses the issues with identifier reuse.

Capability storage

In many real-world systems, processes store capabilities in intermediate capability storage objects. In capability-based systems such as seL4 and Barrelfish, these capability storage objects can be joined together into a chain of storage and shared between different processes, as illustrated in Figure 4.8.

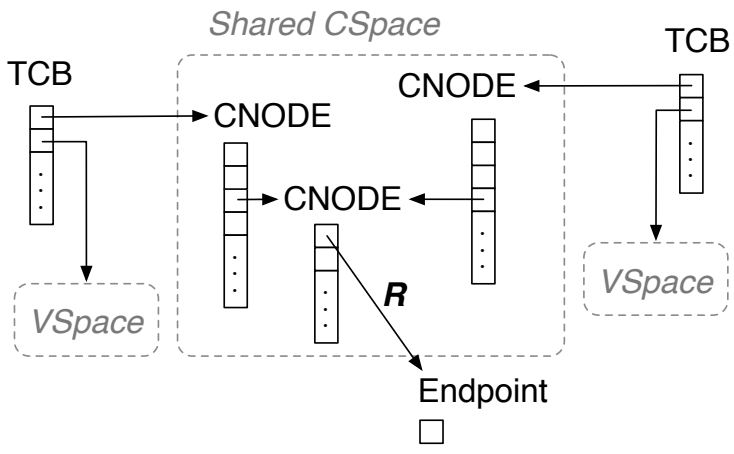


Figure 4.8: A seL4 system with two processes that store their capabilities in a common storage object.

[8]Earlier versions of this work (Boyton, 2009) allowed entity destruction and entity identifier reuse, and tried to sidestep the issue by ensuring that the authority confinement results apply whilst the entities in question have not been destroyed, but this is neither elegant, nor is the property suitable for refinement (as it cannot be expressed as a Hoare triple).

This storage of capabilities by an entity in a (potentially shared) storage object can be modelled as an entity having take and grant rights to the storage object, and copying the capabilities to and from the storage object as required. We believe that this complicates the model as it requires some way of tracking which capabilities are shared, as, for example, when a shared capability is removed, it must be removed from both. Additionally, such a modelling does not correspond closely to how such systems actually work (as they do not do this copying of capabilities). Instead, we here extend the traditional take-grant protection model to make the storage of capabilities in other entities a first-class citizen of the model and allow the use of these capabilities to authorise operations. We introduce a new access right, Store, which confers the authority for an entity to use capabilities stored in another entity as if they were owned by the entity itself. For example, we model the example shown in Figure 4.8 in our take-grant protection model in Figure 4.9. We use two entities Thread 1 and Thread 2 that can store their capabilities in the entities CNode 1 and CNode 2 respectively. Similarly, CNode 1 and CNode 2 are both using CNode 3 as a shared storage entity. This allows Thread 1 to use capabilities located in entities CNode 1 and CNode 3 as its own, and allows either thread to read from the entity Endpoint.

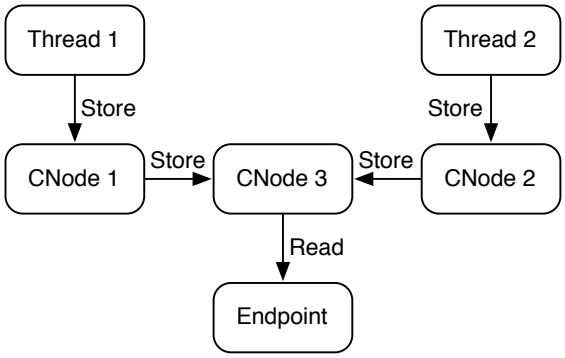


Figure 4.9: Figure 4.8 modelled using Store rights.

We define the capabilities of an entity as the capabilities that they possess directly and capabilities in entities connected via a series of Store rights. Any of these capabilities can be used to authorise an operation performed by an entity. The capability created by an operation can be stored by an entity in any entity connected via a series of Store rights, as illustrated in Figure 4.10. Here, entity e_x is using the Take right stored in the entity $e_{x'}$ to take a copy of entity e_y's Read access to entity e_z (which is stored in entity $e_{y'}$).

To reduce the cumbersome nature of specifying intermediate storage entities in figures, we introduce the notation shown in Figure 4.11 to specify that a capability c is contained either in entity e_o or in an entity connected via a series of Store rights. We define this concept formally in Section 4.3. In the remainder of this

chapter we formally show that this addition of allowing an entity to use capabilities in other store-connected entities does not effect the security of the model.

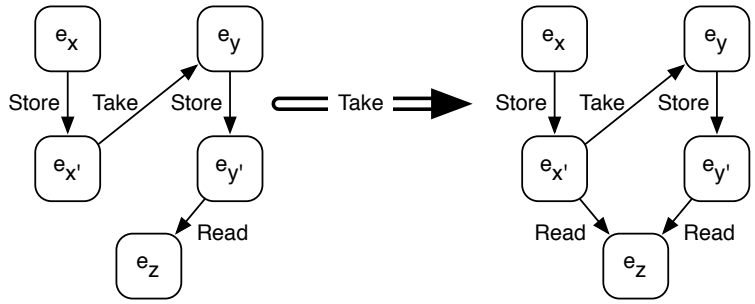


Figure 4.10: An entity e_x, taking a capability from another entity e_y, where both e_x and e_y are storing their capabilities in other entities.

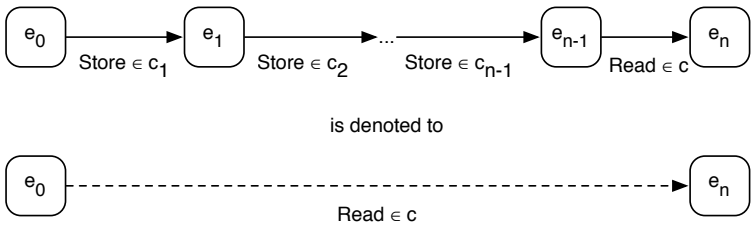


Figure 4.11: Notation for capabilities stored in other entities.

4.3 Formalisation of the state of the take-grant protection model

In this section, we formalise the state of the take-grant protection model in the theorem prover Isabelle/HOL. This model is based on the formalisation by Elkaduwe et al. (2008). In the following sections we formalise the operations and the properties of the model. To formalise the state of the take-grant protection model, we first define precisely the concept of an entity and a capability. In this section, we take a simple state illustrated in Figure 4.12 as a running example to illustrate the definitions.

In our formalisation, each entity has a name associated with it—its *entity_id*. An entity possesses a number of capabilities, where each capability is a set of access rights that the entity has over another entity. The access rights are the standard Read, Write, Take, and Grant, along with the Create and Store access rights

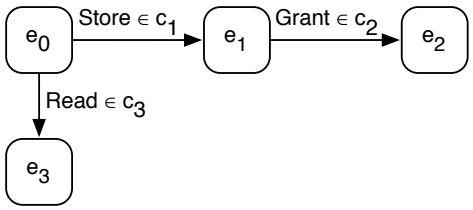


Figure 4.12: An example state. The entity e_0 has capabilities c_1, c_2, and c_3. The capability c_2 is from the *store-connected* entity e_1.

introduced in Section 4.2.[9] We define the access rights, capabilities, and entities formally below.

datatype *right* = Read | Write | Take | Grant | Create | Store
record *cap* = {target :: *entity_id*, rights :: *right set*}

In our example, the capabilities are formalised as:

Definition 4.1 ► *Example capabilities*

$$\begin{aligned} c_1 &\equiv (\!|\, \text{target} = e_1, \text{rights} = \{\text{Store}\} \,|\!) \\ c_2 &\equiv (\!|\, \text{target} = e_2, \text{rights} = \{\text{Grant}\} \,|\!) \\ c_3 &\equiv (\!|\, \text{target} = e_3, \text{rights} = \{\text{Read}\} \,|\!) \end{aligned}$$

The entities of a state are each associated with an identifier, stored as a partial mapping from identifiers to entities, and contain a set of capabilities.

types *state* = *entity_id* ⇒ *entity option*
datatype *entity* = Entity (*cap set*)

We formalise the state, $s_{example}$, of this example as:

Definition 4.2 ► *Example state*

$$s_{example} \equiv [e_0 \mapsto \text{Entity}\ \{c_1, c_3\}, e_1 \mapsto \text{Entity}\ \{c_2\}, e_2 \mapsto \text{Entity}\ \emptyset, e_3 \mapsto \text{Entity}\ \emptyset]$$

We define the predicate is_entity to say when an entity exists at a particular location.

Definition 4.3 ► is_entity

$$\text{is\_entity}\ s\ e \equiv s\ e \neq \text{None}$$

[9] The call right that appeared in the early take-grant work was dropped by Snyder in his 1977 paper (Snyder, 1977), his 1981 paper (Snyder, 1981b), and in his subsequent work. We follow his convention and omit the call right in our work.

One thing that sets this formalisation apart from other take-grant protection models, is the ability for entities to use capabilities stored directly in them and capabilities that are stored in other entities connected via a series of Store rights, as introduced in Section 4.2. The direct capabilities of an entity are the capabilities stored directly within an entity, and are accessed via the function direct_caps_of.

$$\text{direct\_caps\_of } s\ e \equiv \text{case } s\ e \text{ of None} \Rightarrow \emptyset \mid \text{Some } (\text{Entity } caps) \Rightarrow caps$$

◀ **Definition 4.4** direct_caps_of

In our example, the direct capabilities of each of the entities are:

$$\begin{aligned}
&\text{direct_caps_of } \mathsf{s}_{example}\ \mathsf{e}_0 = \{\mathsf{c}_1, \mathsf{c}_3\} \\
&\text{direct_caps_of } \mathsf{s}_{example}\ \mathsf{e}_1 = \{\mathsf{c}_2\} \\
&\text{direct_caps_of } \mathsf{s}_{example}\ \mathsf{e}_2 = \emptyset \\
&\text{direct_caps_of } \mathsf{s}_{example}\ \mathsf{e}_3 = \emptyset
\end{aligned}$$

◀ **Lemma 4.1** *Example direct capabilities*

We define the relation store_connected, which formalises the concept of which entities can store capabilities for another entity. In Figure 4.12, e_0 is store_connected to itself and the entity e_1. The capabilities c_1 and c_3 are direct capabilities of e_0, and c_1, c_2, and c_3 are all capabilities of entity e_0 that can be used to authorise operations.

To define store_connected, we first define the relation store_connected_direct which defines when two entities are connected by a single capability with Store rights, and then define store_connected as the reflexive, transitive closure of the relation store_connected_direct.

$$\begin{aligned}
\text{store_connected_direct } s \equiv \{(e_x, e_y) \mid \exists cap.\ & cap \in \text{direct_caps_of } s\ e_x \wedge \\
& \text{Store} \in \text{rights } cap \wedge \\
& \text{target } cap = e_y\}
\end{aligned}$$

◀ **Definition 4.5** store_connected_direct

$$\text{store\_connected } s \equiv (\text{store\_connected\_direct } s)^*$$

◀ **Definition 4.6** store_connected

In our example, every entity is store_connected to itself, and the entity e_0 is store_connected to the entity e_1.

$$\text{store\_connected } \mathsf{s}_{example} = \{(e_x, e_y) \mid e_x = e_y \vee e_x = \mathsf{e}_0 \wedge e_y = \mathsf{e}_1\}$$

◀ **Lemma 4.2** *Example* store_connected

The capabilities of an entity e are defined as the capabilities stored in any entity that is store_connected to the entity e (including the capabilities in the entity e itself).

Definition 4.7 ► caps_of

$$\text{caps\_of } s\ e \equiv \bigcup (\text{direct\_caps\_of } s \setminus \{e_i \mid (e, e_i) \in \text{store\_connected } s\})$$

In our example, the capabilities of e_0 are therefore c_1, c_2, and c_3, and the only capability of e_1 is c_2.

Lemma 4.3 ► *Example capabilities*

$$\begin{aligned}
&\text{caps_of } s_{example}\ e_0 = \{c_1, c_2, c_3\} \\
&\text{caps_of } s_{example}\ e_1 = \{c_2\} \\
&\text{caps_of } s_{example}\ e_2 = \emptyset \\
&\text{caps_of } s_{example}\ e_3 = \emptyset
\end{aligned}$$

In Figure 4.13 we show the non-Store capabilities of entity e_0 using the store notation of Section 4.2.

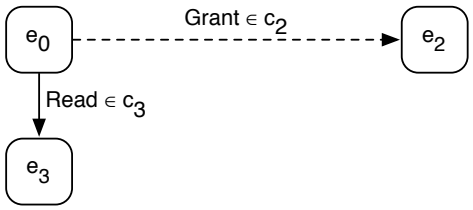


Figure 4.13: Alternate representation of the example in Figure 4.12.

4.4 Operations in the take-grant model

We formalise the standard take-grant protection model operations, namely take, grant, create, and remove, and we extend the model with two new operations revoke and destroy to allow us to more closely model realistic systems. We do this so that this model could be shown to be a formal abstraction of a real-world system such as the seL4 microkernel.

Each operation in the take-grant protection model must be authorised by a capability conferring the necessary access rights. In our formalisation, these access rights can be stored directly or in store_connected entities. Execution in this model is defined using small-step semantics. We model each operation in the take-grant protection model as a system call that is executed by an entity e. We break the checking of authorisation and the execution of the operation into two different stages. A command is executed only if it is legal. Any operation is allowed to non-deterministically fail for reasons not modelled in this level of abstraction, where failure is modelled by an operation not changing the state. The extensions outlined in Section 4.2 (capability storage and restricting entity

creation) and the non-determinism of the model do not have an impact on the access control properties we that show.

Definition 4.8 step

$$\begin{aligned}
&\text{step } cmd\ s \equiv \text{if legal } cmd\ s \text{ then step}'\ cmd\ s \cup \{s\} \text{ else } \{s\} \\
&\quad \textit{where} \\
&\text{step}'\ (\text{SysTake } e\ c_1\ c_2\ R)\ s \equiv \{\text{takeOperation } e\ c_1\ c_2\ R\ s\} \\
&\text{step}'\ (\text{SysGrant } e\ c_1\ c_2\ R)\ s \equiv \{\text{grantOperation } e\ c_1\ c_2\ R\ s\} \\
&\text{step}'\ (\text{SysCreate } e\ c_1\ c_2)\ s \equiv \{\text{createOperation } e\ c_1\ c_2\ s\} \\
&\text{step}'\ (\text{SysRemove } e\ c_1\ c_2)\ s \equiv \{\text{removeOperation } e\ c_1\ c_2\ s\} \\
&\text{step}'\ (\text{SysRevoke } e\ c)\ s \equiv \text{revokeOperation } e\ c\ s \\
&\text{step}'\ (\text{SysDestroy } e\ c)\ s \equiv \{\text{destroyOperation } e\ c\ s\} \\
&\text{step}'\ (\text{SysCopy } e\ c_1\ c_2\ R)\ s \equiv \{\text{copyOperation } e\ c_1\ c_2\ R\ s\}
\end{aligned}$$

We define execution in our model as the execution of a list of commands from right to left.

Definition 4.9 execute

$$\begin{aligned}
&\text{execute } []\ s = \{s\} \\
&\text{execute } (cmd{\cdot}cmds)\ s = \bigcup (\text{step } cmd \setminus \text{execute } cmds\ s)
\end{aligned}$$

The legal checks are shown in Figure 4.14, and the definitions of all of the operations are shown below.

$$\begin{aligned}
&\text{legal } (\text{SysTake } e\ c_1\ c_2\ r)\ s \equiv \text{is_entity } s\ e \wedge \text{is_entity } s\ (\text{target } c_1) \wedge \\
&\qquad c_1 \in \text{caps_of } s\ e \wedge c_2 \in \text{caps_of } s\ (\text{target } c_1) \wedge \text{Take} \in \text{rights } c_1 \\
&\text{legal } (\text{SysGrant } e\ c_1\ c_2\ r)\ s \equiv \text{is_entity } s\ e \wedge \text{is_entity } s\ (\text{target } c_1) \wedge \\
&\qquad \{c_1, c_2\} \subseteq \text{caps_of } s\ e \wedge \text{Grant} \in \text{rights } c_1 \\
&\text{legal } (\text{SysCreate } e\ c_1\ c_2)\ s \equiv \text{is_entity } s\ e \wedge \text{is_entity } s\ (\text{target } c_1) \wedge \neg\ \text{is_entity } s\ (\text{target } c_2) \wedge \\
&\qquad \{c_1, c_2\} \subseteq \text{caps_of } s\ e \wedge \\
&\qquad \text{Write} \in \text{rights } c_1 \wedge \text{Store} \in \text{rights } c_1 \wedge \text{Create} \in \text{rights } c_2 \\
&\text{legal } (\text{SysRemove } e\ c_1\ c_2)\ s \equiv \text{is_entity } s\ e \wedge c_1 \in \text{caps_of } s\ e \\
&\text{legal } (\text{SysRevoke } e\ c)\ s \equiv \text{is_entity } s\ e \wedge c \in \text{caps_of } s\ e \\
&\text{legal } (\text{SysDestroy } e\ c)\ s \equiv \text{is_entity } s\ e \wedge c \in \text{caps_of } s\ e \wedge \{\text{Create}\} = \text{rights } c \wedge \\
&\qquad \text{target } c \notin \text{target} \setminus (\text{all_caps_of } s - \{c\}) \\
&\text{legal } (\text{SysCopy } e\ c_1\ c_2\ r)\ s \equiv \text{is_entity } s\ e \wedge \text{is_entity } s\ (\text{target } c_1) \wedge \\
&\qquad \{c_1, c_2\} \subseteq \text{caps_of } s\ e \wedge \text{Store} \in \text{rights } c_1
\end{aligned}$$

Figure 4.14: Definition of when an operation is legal to perform.

Take An entity e with a capability c_1 with a Take right to another entity e_1 is able to take a (potentially diminished) copy of that entity's capabilities.

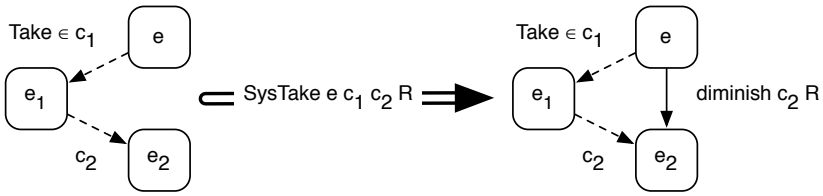


Figure 4.15: Take Operation

takeOperation

Definition 4.10 ►

$$\text{takeOperation } e\ c_1\ c_2\ R\ s \equiv s(e \mapsto \text{Entity } (\{\text{diminish } R\ c_2\} \cup \text{direct\_caps\_of } s\ e))$$

where $\text{diminish } R\ cap \equiv cap(\!|\ \text{rights} := \text{rights } cap \cap R\ |\!)$

Grant An entity e with a capability c_1 with a Grant right to an entity e_1 is able to give a (potentially diminished) copy of one of its capabilities to that entity.

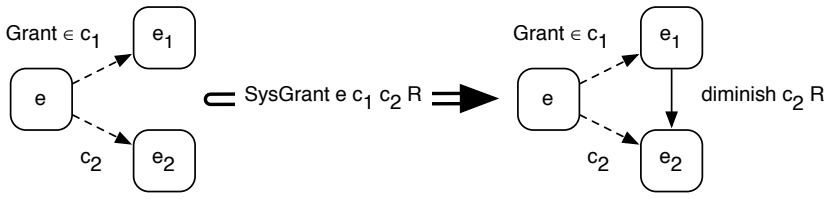


Figure 4.16: Grant Operation

grantOperation

Definition 4.11 ►

$$\text{grantOperation } e\ c_1\ c_2\ R\ s \equiv$$
$$s(\text{target } c_1 \mapsto \text{Entity } (\{\text{diminish } R\ c_2\} \cup \text{direct\_caps\_of } s\ (\text{target } c_1)))$$

Create An entity e possessing a Create capability c_2 to an identifier e_2 that is presently unused by any entity can create a new entity with identifier e_2 in this location. This is a modification of the standard take-grant models where every entity has implicit permissions to create new entities, as explained in Section 4.2.

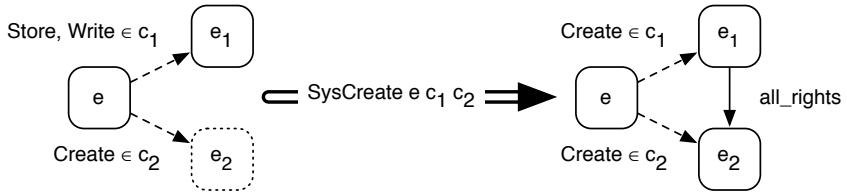


Figure 4.17: Create Operation

◀ **Definition 4.12** createOperation

```
createOperation e c1 c2 s ≡
 let new_cap = (| target = target c2, rights = all_rights |);
     new_target = {new_cap} ∪ direct_caps_of s (target c1);
     new_entity = Entity ∅
 in s(target c1 ↦ Entity new_target, target c2 ↦ new_entity)
 where  all_rights ≡ {Read, Write, Take, Grant, Create, Store}
```

Remove An entity e that possess a capability c_1 to another entity e_1, is able to remove any capability c_2 from that entity.

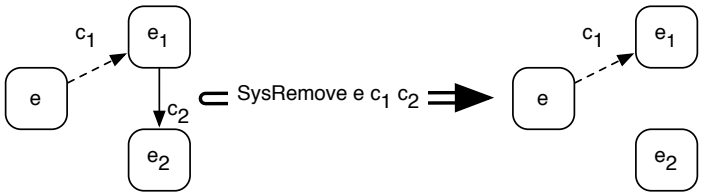


Figure 4.18: Remove Operation

◀ **Definition 4.13** removeOperation

```
removeOperation e c1 c2 s ≡
 case s (target c1) of None ⇒ s
  | Some (Entity caps) ⇒ s(target c1 ↦ Entity (caps − {c2}))
```

Revoke In some capability systems such as seL4, an entity is able to revoke access to all capabilities that have been derived[10] from a capability it possesses. For example, if an entity e has granted a copy of a capability it possesses c_1 to another entity e_2, then it can later revoke the access from the entity e_2. The seL4 microkernel tracks the derivation of capabilities through a *capability derivation tree* (Elkaduwe et al., 2007). In our formalism, which is designed to be a formal abstraction of capability systems such as seL4, we model revocation of a capability c as a non-deterministic deletion of any capabilities in any entities e_i pointing to the same entity that the capability c does. We do this by constructing a function, *cap_map*, mapping from entity identifiers to the set of capabilities to delete from them (which is a subset of the capabilities that point to the same entity as the capability c does).

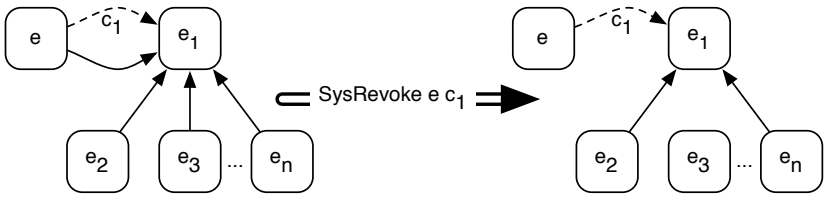


Figure 4.19: Revoke Operation

Definition 4.14 ► revokeOperation

revokeOperation $e\ c\ s \equiv$
$\{s' \mid \exists cap\_map.\ \forall e_i.\ cap\_map\ e_i \subseteq \{cap \in \text{direct\_caps\_of}\ s\ e_i \mid \text{target}\ cap = \text{target}\ c\} \wedge$
$s' = \text{removeSetOfCaps}\ cap\_map\ s\}$
where removeSetOfCaps $cap\_map\ s \equiv$
$\lambda e.\ \text{case}\ s\ e\ \text{of None} \Rightarrow \text{None}$
$\mid \text{Some}\ (\text{Entity}\ caps) \Rightarrow \text{Some}\ (\text{Entity}\ (caps - cap\_map\ e))$

[10] That is, capabilities that were created as a copy of another capability.

Destroy To model real-world systems, we add a destroy operation to the formal model to allow the removal of entities. This operation is the counterpart to the create operation we introduced earlier.[11] An entity e may destroy another entity e_1 if it possesses a capability c with Create rights to the entity, and if no other capabilities to the entity e_1 exist.

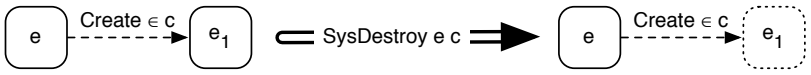


Figure 4.20: Destroy Operation

$$\text{destroyOperation } e\ c\ s \equiv s(\text{target } c := \text{None})$$

◀ **Definition 4.15** destroyOperation

Copy As we have extended the standard take-grant model to allow the storage of capabilities in store_connected entities, we add the copy operation to allow an entity e to copy one of its capabilities c_2 into an entity e_1 to which it possesses a capability c_1 with Store rights. This allows an entity to manage the storage of its capabilities.

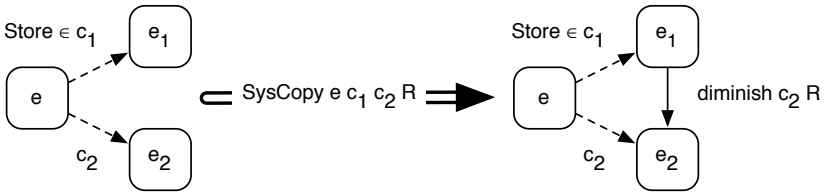


Figure 4.21: Copy Operation

$$\text{copyOperation } sRef\ c_1\ c_2\ R\ s \equiv$$
$$s(\text{target } c_1 \mapsto \text{Entity } (\{\text{diminish } R\ c_2\} \cup \text{direct\_caps\_of } s\ (\text{target } c_1)))$$

◀ **Definition 4.16** copyOperation

4.5 Authority confinement

The key result of the take-grant protection model is that the security question, "is it true that an entity e_x can α an entity e_y?" is decidable (where α can be any access right, such as Read or Write). Looking at the state shown in Figure 4.22, we can see that the answer to the question "can entity e_1 gain explicit read access to entity e_2?" is *yes*.

[11] The destroy operation is not part of the standard take-grant protection model, but it was added in the diminished-take model (Shapiro, 1999) and is present in the HRU formalisation (Harrison et al., 1976).

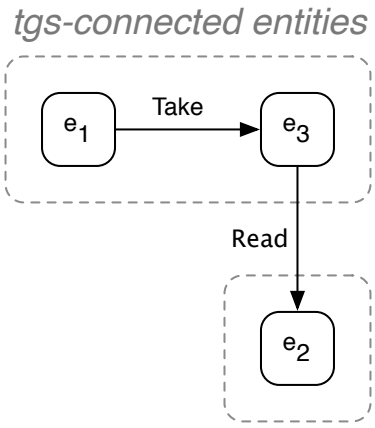


Figure 4.22: In this example, entity e_1 can gain read access to entity e_2.

As explained in Section 4.1, an entity with the ability to create new entities can bypass the directionality of Take and Grant access rights. Because of this, when examining how capabilities can propagate through a system, it does not make sense to examine to examine the directionality of Take and Grant access rights, but instead to examine when two entities are connected by capabilities with either Take or Grant access rights in either direction. Lipton and Snyder (1977) show in this standard take-grant protection model, entities can transfer capabilities if and only if they are connected though a series of capabilities with Take or Grant access rights. Such entities are called *tg-connected*.[12]

With the addition of the Store access right, we extend the existing literature by stating that entities can transfer capabilities if they are *tgs-connected*, that is, if they are connected by a series of capabilities with Take, Grant, or Store access rights. We also say that if an entity e_x possesses a capability with Create rights to another entity e_y, then entity e_x possesses full access to entity e_y. We justify both of these changes below.

Firstly, given that an entity can copy a capability from any store_connected entity to any entity that it possesses a store capability to, we need to include Store rights in the definition of *connected*.

Secondly, as explained in detail in Section 4.2, if an entity e_x has Create rights to another entity e_y, then (assuming no other entity has any access rights to the entity e_y) entity e_x may destroy entity e_y and recreate it, giving entity e_x full access rights to entity e_y. Because of this, for the purposes of analysing the transfer of capabilities, we defined a function extra_rights that adds the implicit full access rights that an entity with Create rights to another entity has to that entity.

[12]Given that we can, in our model, restrict the creation of entities, we could tighten our definition of *tg-connected* to be directional for systems with sufficiently restricted entity creation, but in this model we focus on primarily on more dynamic systems.

$$\mathsf{extra\_rights}\ cap \equiv \mathsf{if}\ \mathsf{Create} \in \mathsf{rights}\ cap\ \mathsf{then}\ cap(\!|\ \mathsf{rights} := \mathsf{all\_rights}\ |\!)\ \mathsf{else}\ cap$$

◀ **Definition 4.17**
extra_rights

In the remainder of this section we will formalise the question we started the section with—can an entity e_x gain specific access rights to an entity e_y? We formalise this by answering the question "can an entity e_x gain more access to an entity e_y than what the capability c possesses?". We show that an entity e_x can gain at most as much access to an entity e_y as what any entity tgs_connected to entity e_x possesses. Thus, if no entity tgs_connected to entity e_x possesses Write rights to entity e_y, then entity e_x cannot acquire such rights (even if other disconnected entities possess these rights). Formally, we express this result in Lemma 4.4 as we explain below.

We say that in any future state s', after a number of commands *cmds* have executed ($s' \in \mathsf{execute}\ cmds\ s$), then an entity e_x can have at most as much authority to an object as is possessed by the capability c ($\mathsf{caps\_of}\ s'\ e_x \trianglelefteq_{\mathsf{cap}} cap$), if all the entities e_i which are tgs_connected also only have as much authority to an object as the capability *cap* does ($\forall e_i.\ s \vdash e_x \leftrightarrow^* e_i \longrightarrow \mathsf{caps\_of}\ s\ e_i \trianglelefteq_{\mathsf{cap}} cap$). We introduce these concepts formally below.

$$\frac{s' \in \mathsf{execute}\ cmds\ s \qquad \forall e_i.\ s \vdash e_x \leftrightarrow^* e_i \longrightarrow \mathsf{caps\_of}\ s\ e_i \trianglelefteq_{\mathsf{cap}} cap}{\mathsf{caps\_of}\ s'\ e_x \trianglelefteq_{\mathsf{cap}} cap}$$

◀ **Lemma 4.4**
Authority confinement

We first define notation for capabilities with read, write, take, grant, create, and store access to an entity e.

$$\begin{aligned}
\mathsf{read_cap}\ e &\equiv (\!|\ \mathsf{target} = e, \mathsf{rights} = \{\mathsf{Read}\}\ |\!)\\
\mathsf{write_cap}\ e &\equiv (\!|\ \mathsf{target} = e, \mathsf{rights} = \{\mathsf{Write}\}\ |\!)\\
\mathsf{take_cap}\ e &\equiv (\!|\ \mathsf{target} = e, \mathsf{rights} = \{\mathsf{Take}\}\ |\!)\\
\mathsf{grant_cap}\ e &\equiv (\!|\ \mathsf{target} = e, \mathsf{rights} = \{\mathsf{Grant}\}\ |\!)\\
\mathsf{create_cap}\ e &\equiv (\!|\ \mathsf{target} = e, \mathsf{rights} = \{\mathsf{Create}\}\ |\!)\\
\mathsf{store_cap}\ e &\equiv (\!|\ \mathsf{target} = e, \mathsf{rights} = \{\mathsf{Store}\}\ |\!)\\
\mathsf{full_cap}\ e &\equiv (\!|\ \mathsf{target} = e, \mathsf{rights} = \mathsf{all_rights}\ |\!)
\end{aligned}$$

An entity contains a set of capabilities, with each capability possessing a set of access rights to a specific entity. We define $caps \trianglelefteq_{\mathsf{cap}} cap$ to say that the authority that a set of capabilities *caps* has to an entity is dominated by that of the capability *cap* if the capability *cap* has more (or equal) authority to the entity *cap* points to than what any of the capabilities in the set *caps* do (where we say that possessing Create rights implies having full access).

Definition 4.18 ► *Capabilities dominated by a capability*

$$caps \trianglelefteq_{\mathsf{cap}} cap \equiv$$
$$\forall cap' \in caps.\ \mathsf{target}\ cap' = \mathsf{target}\ cap \longrightarrow$$
$$\mathsf{rights}\ (\mathsf{extra\_rights}\ cap') \subseteq \mathsf{rights}\ (\mathsf{extra\_rights}\ cap)$$

For example, the authority possessed by a set of capabilities a consisting of a read_cap and write_cap to an entity e_x is *not* dominated by that of a take_cap to an entity e_x (that is, $\neg\ \{\mathsf{read\_cap}\ e_x, \mathsf{write\_cap}\ e_x\} \trianglelefteq_{\mathsf{cap}} \mathsf{take\_cap}\ e_x$). The authority possessed by a read_cap and a write_cap to an entity e_x *is* dominated by that of a create_cap to an entity e_x (that is, $\{\mathsf{read\_cap}\ e_x, \mathsf{write\_cap}\ e_x\} \trianglelefteq_{\mathsf{cap}} \mathsf{create\_cap}\ e_x$) as possessing Create access rights to an entity implies full assess rights to the entity.

Similarly, we define $cap \in_{\mathsf{cap}} caps$ to say that there is a capability in the set *caps* that has at least as much authority to an entity as the capability *cap* has.

Definition 4.19 ► *Capability rights in capabilities*

$$cap \in_{\mathsf{cap}} caps \equiv$$
$$\exists cap' \in caps.\ \mathsf{target}\ cap = \mathsf{target}\ cap' \wedge$$
$$\mathsf{rights}\ (\mathsf{extra\_rights}\ cap) \subseteq \mathsf{rights}\ (\mathsf{extra\_rights}\ cap')$$

For example, the authority possessed by a take_cap to an entity e_x is not possessed by the set of capabilities containing a read_cap e_x and a write_cap e_x (that is, $\mathsf{take\_cap}\ e_x \notin_{\mathsf{cap}} \{\mathsf{read\_cap}\ e_x, \mathsf{write\_cap}\ e_x\}$), but the authority possessed by a take_cap to an entity e_x is possessed by the set of capabilities containing a read_cap and a create_cap (that is, $\mathsf{take\_cap}\ e_x \in_{\mathsf{cap}} \{\mathsf{read\_cap}\ e_x, \mathsf{create\_cap}\ e_x\}$).

To define tgs_connected, we first define the concept of an entity e_x being able to leak capabilities to another entity e_y if entity e_y has a capability with Take rights to entity e_x, if entity e_x has a capability with Grant rights to entity e_y, or if they can both store capabilities in a common entity e_i (by both being store_connected to a mutual entity e_i).

Definition 4.20 ► leak

$$s \vdash e_x \rightarrow e_y \equiv \mathsf{take\_cap}\ e_x \in_{\mathsf{cap}} \mathsf{caps\_of}\ s\ e_y \vee \mathsf{grant\_cap}\ e_y \in_{\mathsf{cap}} \mathsf{caps\_of}\ s\ e_x \vee$$
$$(\exists e_i.\ (e_x, e_i) \in \mathsf{store\_connected}\ s \wedge (e_y, e_i) \in \mathsf{store\_connected}\ s)$$

We define the relation directly_tgs_connected as the reflexive closure of leak, and tgs_connected as the reflexive, symmetric, transitive closure of leak.[13]

$$s \vdash e_x \leftrightarrow e_y \equiv s \vdash e_x \rightarrow e_y \vee s \vdash e_y \rightarrow e_x$$

◀ **Definition 4.21** directly_tgs_connected

$$s \vdash e_x \leftrightarrow^* e_y \equiv (e_x, e_y) \in \{(e_x, e_y) \mid s \vdash e_x \leftrightarrow e_y\}^*$$

◀ **Definition 4.22** tgs_connected

Using these definitions, we first prove that none of our operations can ever connect disconnected entities. If two entities are transitively connected after a sequence of operations, they must have been transitively connected before, as formalised in Theorem 4.1.

A generalised operation

Rather than proving individual properties about the operations takeOperation, grantOperation, copyOperation, and createOperation, we simplify our analysis by showing that each of these operations can be considered to be specialisations of a more general operation generalOperation (where createOperation also creates a new entity).

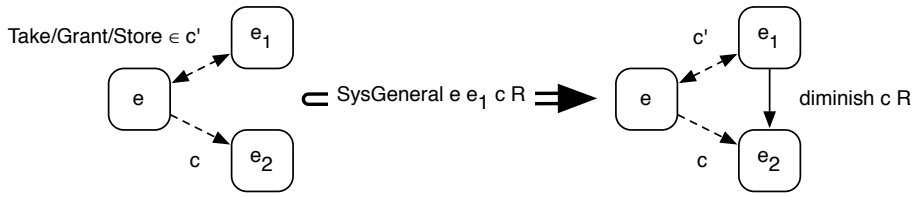


Figure 4.23: Generalised capability creation operation.

generalOperation $e\ e_1\ c\ r\ s \equiv$
$s(e_1 \mapsto \text{Entity}\ (\{\text{diminish}\ r\ (\text{extra\_rights}\ c)\} \cup \text{direct\_caps\_of}\ s\ e_1))$

◀ **Definition 4.23** generalOperation

[13] We have also shown that this definition is equivalent to the transitive closure of Take, Grant, and Store (hence the name). The definition we present here more closely models the concept of shared storage, which is why we use it.

Lemma 4.5 ▶ *General operation rewrites*

$$\begin{aligned}
&\mathsf{takeOperation}\ e\ c_1\ c_2\ r\ s &&= \mathsf{generalOperation}\ (\mathsf{target}\ c_1)\ e\ c_2\ (r \cap \mathsf{rights}\ c_2)\ s\\
&\mathsf{grantOperation}\ e\ c_1\ c_2\ r\ s &&= \mathsf{generalOperation}\ e\ (\mathsf{target}\ c_1)\ c_2\ (r \cap \mathsf{rights}\ c_2)\ s\\
&\mathsf{copyOperation}\ e\ c_1\ c_2\ r\ s &&= \mathsf{generalOperation}\ e\ (\mathsf{target}\ c_1)\ c_2\ (r \cap \mathsf{rights}\ c_2)\ s\\
&\text{If } \mathsf{legal}\ (\mathsf{SysCreate}\ e\ c_1\ c_2)\ s \text{ then}\\
&\mathsf{createOperation}\ e\ c_1\ c_2\ s &&= \mathsf{generalOperation}\ e\ (\mathsf{target}\ c_1)\ c_2\ \mathsf{all_rights}\ s(\mathsf{target}\ c_2 \mapsto \mathsf{null_entity})
\end{aligned}$$

Proof: Each of these definitions can be shown by expanding the definitions of the operations. The createOperation can be composed of a generalOperation followed by updating the state to add the empty entity at the location pointed to by the capability c_2, provided that the capability c_2 possesses Create access rights, which is true if the createOperation is legal. ■

These operations are legal to execute only if the entity e and the target entity pointed to by the capability c_1 (namely target c_1) are already directly_tgs_connected.

Lemma 4.6 ▶ *General operation rewrites*

$$\frac{\mathsf{legal}\ (\mathsf{SysTake}\ e\ c_1\ c_2\ r)\ s}{s \vdash e \leftrightarrow \mathsf{target}\ c_1} \qquad \frac{\mathsf{legal}\ (\mathsf{SysGrant}\ e\ c_1\ c_2\ r)\ s}{s \vdash e \leftrightarrow \mathsf{target}\ c_1}$$

$$\frac{\mathsf{legal}\ (\mathsf{SysCopy}\ e\ c_1\ c_2\ r)\ s}{s \vdash e \leftrightarrow \mathsf{target}\ c_1} \qquad \frac{\mathsf{legal}\ (\mathsf{SysCreate}\ e\ c_1\ c_2)\ s}{s \vdash e \leftrightarrow \mathsf{target}\ c_1}$$

Proof: Each of the operations are legal only if the entity e and the entity pointed to by the capability c_1 are connected by a capability with either Take, Grant, Store, or Create access rights respectively, all of which imply the two entities e_x and target c_1 are directly_tgs_connected. ■

This generalised operation does not tgs-connect any entities that are not already tgs_connected. To show this, we first show that if any two entities e_x and e_y are directly_tgs_connected after a general operation, then they must have been tgs_connected before, as shown in Lemma 4.7.

Lemma 4.7 ▶ *General operation connected*

$$\frac{\mathsf{generalOperation}\ e\ e_1\ cap\ r\ s \vdash e_x \leftrightarrow e_y \qquad cap \in \mathsf{caps\_of}\ s\ e \qquad s \vdash e \leftrightarrow e_1}{s \vdash e_x \leftrightarrow^* e_y}$$

Proof: This is true as either e_x and e_y are already directly_tgs_connected, or they are connected by the new capability added (diminish c R), which happens only when they are already tgs_connected via entity e. ■

As the other operations of the model, SysRemove, SysRevoke, and SysDestroy, remove either capabilities or entities, we can show that no operation directly tgs-connects any entities that are not already tgs_connected.

$$\frac{s' \in \text{step } cmd\ s \qquad s' \vdash e_x \leftrightarrow e_y}{s \vdash e_x \leftrightarrow^* e_y}$$

◀ **Lemma 4.8**
Connected was directly connected

Proof: By Lemma 4.7, we know that the commands SysTake, SysGrant, SysCopy, and SysCreate do not connect disconnected entities. Since the other operations (SysRemove, SysRevoke, and SysDestroy) do not create new capabilities they also cannot connect disconnected entities. ■

By induction, we extend Lemma 4.8 to show that if two entities e_x and e_y are tgs_connected after the execution of a sequence of commands, then they must have been previously tgs_connected. We note that this result is true, even when we allow for entity destruction within this series of commands.

$$\frac{s' \in \text{execute } cmds\ s \qquad s' \vdash e_x \leftrightarrow^* e_y}{s \vdash e_x \leftrightarrow^* e_y}$$

◀ **Theorem 4.1**
tgs_connected *is preserved*

Proof: This is proven by induction, first on the sequence of commands *cmds*, and then on the reflexive, transitive closure of the tgs_connected relation and using Lemma 4.8. ■

A corollary of this theorem is that if entities are not tgs_connected, then they cannot leak authority.

$$\frac{s' \in \text{execute } cmds\ s \qquad \neg\, s \vdash e_x \leftrightarrow^* e_y}{\neg\, s' \vdash e_x \rightarrow e_y}$$

◀ **Corollary 4.1**
Non-connected will not leak

We now show that the capabilities attainable by an entity are only those that already exist in a tgs_connected entity, formalising the question we started the section with. First, we show that, if a state s' is the result of an operation *cmd* on the state s, and a capability *cap* is present in the capabilities of an entity e_x, then this capability must have come from an entity e_i which is tgs_connected to the entity e_x, which itself possesses the authority that the capability *cap* confers (noting that possessing Create rights to an entity implies possessing all rights to that entity).

Lemma 4.9 ► *Capabilities after an operation*

$$\frac{s' \in \mathsf{step}\ cmd\ s \qquad cap \in \mathsf{caps\_of}\ s'\ e_x}{\exists e_i.\ s \vdash e_x \leftrightarrow^* e_i \wedge cap \in_{\mathsf{cap}} \mathsf{caps\_of}\ s\ e_i}$$

Proof: We divide the operations into two groups for analysis—those that can be expressed using generalOperation and those that remove either capabilities or entities (that is, SysRemove, SysRevoke, and SysDestroy).

The lemma can be seen to be true for the former operations by noting that any new capabilities created by generalOperation either existed already, or were present in a connected entity (where possessing a capability with Create access rights to an entity implies possessing full access rights to that entity).

The other operations remove either capabilities or entities, and so any capabilities present in an entity after these operations must have been present already in the entity itself. ■

Using this, we show authority confinement for our model. We say that the capabilities of an entity e_x to an entity are dominated by a capability *cap* in any future state s' if, in our initial state s, the capabilities in all the entities that are tgs_connected to the entity e_x are also dominated by the capability *cap*.

Theorem 4.2 ► *Authority confinement*

$$\frac{s' \in \mathsf{execute}\ cmds\ s \qquad \forall e_i.\ s \vdash e_x \leftrightarrow^* e_i \longrightarrow \mathsf{caps\_of}\ s\ e_i \trianglelefteq_{\mathsf{cap}} cap}{\mathsf{caps\_of}\ s'\ e_x \trianglelefteq_{\mathsf{cap}} cap}$$

Proof: To prove this, we first strengthen the induction hypothesis to:

$$\frac{s' \in \mathsf{execute}\ cmds\ s \qquad \forall e_i.\ s \vdash e_x \leftrightarrow^* e_i \longrightarrow \mathsf{caps\_of}\ s\ e_i \trianglelefteq_{\mathsf{cap}} cap \qquad s' \vdash e_x \leftrightarrow^* e_i}{\mathsf{caps\_of}\ s'\ e_i \trianglelefteq_{\mathsf{cap}} cap}$$

We then induct on the commands *cmds*, and use Lemma 4.9 and the first inductive step of Theorem 4.1. ■

4.6 Isolation

Snyder (1981b) and later Bishop (1996) use the reflexive, symmetric, transitive nature of the tgs_connected relation to define equivalence classes of entities, which they call *islands*. Using this concept, we can speak of the authority possessed by an island of entities to a specific entity, and use this to express the isolation property from Section 4.5.

$$\text{island } s\ e_x \equiv \{e_i \mid s \vdash e_x \leftrightarrow^* e_i\}$$

◀ **Definition 4.24** island

Since no entity e_x within an island can acquire more access to another entity e_y (either within or external to that island) than what any other entity within an island can, it makes sense to define the concept of the capabilities that an island of entities possess. We define this as the capabilities possessed by entities within that island.[14]

$$\text{island\_caps } s\ e_x \equiv \bigcup (\text{caps\_of } s \setminus \text{island } s\ e_x)$$

◀ **Definition 4.25** island_caps

Using this definition, we reformulate the final authority confinement statement from Section 4.5 to say that the authority of an island does not increase. First we show simpler way of calculating the capabilities of an island.

$$(\text{island\_caps } s\ e_x \trianglelefteq_{\text{cap}} cap) = (\forall e_i.\ s \vdash e_x \leftrightarrow^* e_i \longrightarrow \text{caps\_of } s\ e_i \trianglelefteq_{\text{cap}} cap)$$

◀ **Lemma 4.10** *Island caps are bound*

Proof: This is shown by expanding the definitions of island, island_caps, and $caps \trianglelefteq_{\text{cap}} cap$. ■

We use this to show a reformulation of Theorem 4.2 in terms of the authority of islands.

$$\frac{s' \in \text{execute } cmds\ s \qquad \text{island\_caps } s\ e_x \trianglelefteq_{\text{cap}} cap}{\text{island\_caps } s'\ e_x \trianglelefteq_{\text{cap}} cap}$$

◀ **Theorem 4.3** *Authority confinement of Islands*

Proof: The capabilities of an island are those stored in tgs_connected entities. We know by Theorem 4.1 that any entities that are tgs_connected in the future were tgs_connected previously. Then, using Lemma 4.10, we rewrite this theorem into the form of Theorem 4.2. ■

4.7 Information flow

Using the authority confinement proofs, we now examine the information flow channels that are possible using the authority confinement graph of a system. For

[14] It makes no difference whether we examine the capabilities or the direct capabilities of the entities within an island, as these are the same. We have proven using our above definition that $\text{island\_caps } s\ e_x = \bigcup (\text{direct\_caps\_of } s \setminus \text{island } s\ e_x)$.

example, taking the state in Figure 4.24 and assuming the required entities work together, we can see that information can flow from the island with entity e_1 to the island with entity e_2, and from the island with entity e_2 to the island with entity e_3, despite the fact that no entity in the island with entity e_1 has any capabilities to any of the entities in the island with entity e_3 (nor vice versa). Formally, we say that information can flow from one island to another if the former can write to the latter, or if the latter can read from the former. This analysis is similar to the de facto rights first analysed by Bishop and Snyder (1979).

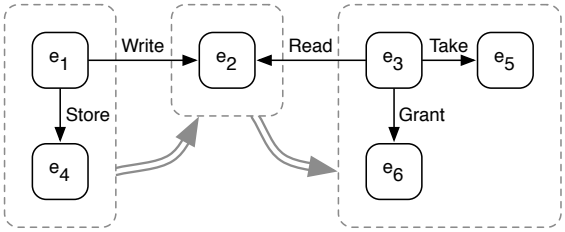


Figure 4.24: In this example, information can flow from the island with entity e_1 to the island with entity e_2, and from the island with entity e_2 to the island with entity e_3.

Definition 4.26 ► flow

$$s \vdash e_x \rightsquigarrow e_y \equiv \exists e_{x'} \in \mathsf{island}\ s\ e_x.\ \exists e_{y'} \in \mathsf{island}\ s\ e_y.\ \mathsf{read\_cap}\ e_{x'} \in_{\mathsf{cap}} \mathsf{caps\_of}\ s\ e_{y'} \vee \mathsf{write\_cap}\ e_{y'} \in_{\mathsf{cap}} \mathsf{caps\_of}\ s\ e_{x'}$$

We define transitive information flow as the reflexive, transitive (but not symmetric) flow of information.

Definition 4.27 ► *Transitive* flow

$$s \vdash e_x \rightsquigarrow^* e_y \equiv (e_x, e_y) \in \{(e_x, e_y) \mid s \vdash e_x \rightsquigarrow e_y\}^*$$

Given this definition of information flow, we then show that, since the capabilities of an island do not increase, the information flow between two islands also does not increase.

Theorem 4.4 ► *Information flow*

$$\frac{s' \in \mathsf{execute}\ cmds\ s \qquad s' \vdash e_x \rightsquigarrow^* e_y}{s \vdash e_x \rightsquigarrow^* e_y}$$

Proof: First, we induct over the list of commands, and are left needing to prove the result for a single step of execution.

Then, in similar way to Theorem 4.1, we induct on the reflexive, transitive closure of the $s \vdash e_x \rightsquigarrow^* e_y$ relation. The base case is obviously true. In the inductive case, we have two islands, island e_i and island e_j, where information can flow from the former to the latter. This means that there are entities $e_{i'}$ and $e_{j'}$ in these

respective islands such that either entity $e_{i'}$ has a write capability to entity $e_{j'}$, or entity $e_{j'}$ has a read capability to entity $e_{i'}$.

By the first inductive step of Theorem 4.1, we know that if the entity $e_{i'}$ is in island e_i after a series of operations, then it must have been in the island beforehand (and similarly with entity $e_{j'}$ and island e_j).

Since by Lemma 4.9 we know that any access rights after an operation must have existed in a tgs_connected entity previously, we know that the capability has to be present in the island originally. ■

A corollary of this result is that if information cannot flow between islands initially, then it can never flow, despite the fact that the islands themselves might split and change.

$$\frac{s' \in \text{execute } cmds\ s \qquad \neg\, s \vdash e_x \rightsquigarrow^* e_y}{\neg\, s' \vdash e_x \rightsquigarrow^* e_y}$$

◀ **Corollary 4.2**
No information flow

4.8 Conclusion

In this chapter we have presented a formalisation of the take-grant protection model in Isabelle/HOL, based on that of Elkaduwe et al. (2008), and extended it with the addition of shared capability storage as a first class citizen, and an explicit create right to be able to create an entity using a particular identifier.

Our changes to entity creation more closely model the behaviour of the seL4 microkernel and the finite nature of memory than previous models.

Our formalisation allows authorisation of operations performed by an entity by capabilities stored in other entities. This does not change the security implications of the model, as such entities would be able to copy such capabilities to be stored directly within the entity, but it models the actions of an operating system such as seL4, which stores capabilities in intermediate objects more closely.

To our knowledge, allowing identifier reuse through our explicit create right conferring the permission to be able to create an entity using a particular identifier, and the addition shared capability storage as a first class citizen, are new contributions to the take-grant protection model. We have shown that these additions do not change either the de jure or the de facto security properties of the take-grant protection model.

Chapter Summary

- The take-grant protection model is a family of formal models where security questions such as "can the entity x ever gain read access to the entity y?" are decidable.
- These models consists of *entities* which possess *capabilities* to other entities. Entities can *take* capabilities from and *grant* capabilities to other entities that they have Take or Grant capabilities to, respectively. In traditional take-grant protection models, entities can also *create* an unbounded number of new entities.
- We have extended the traditional take-grant protection models with the addition of shared capability storage as a first class citizen and an explicit Create right to be able to create an entity using a particular identifier. These additions more closely model the behaviour of real-world systems such as the seL4 microkernel, and do not change the security results of the model.
- There are two main security questions analysed of the take-grant protection models: the authority an entity can *explicitly* gain access to and the information an entity could *implicitly* gain access to.
- Entities are able to gain any capabilities possessed by any other entities connected by a series of Take, Grant, or Store capabilities.
- We can divide entities into *islands* of entities connected by Take, Grant, and Store capabilities. We can then analyse the way *capabilities* and *information* can flow between these islands.

Chapter 5

System initialisation

5.1 Initialisation of computer systems

The initialisation of computer systems is a hard problem. The abstractions commonly used to configure systems are generally in the process of being initialised themselves and thus cannot be used. Additionally, the initial processes that configure a system are often given elevated authority and special care must be taken to ensure that these processes appropriately manage and diminish this authority (Shapiro and Hardy, 2002).

There has been a lot of recent work on ensuring that the code running on a system is in fact the correct code. Trusted Platform Modules (TPMs) and code signing are both used to enforce that each level of initialisation code is in fact the correct code (Kauer, 2007). Systems are often built up in a layered approach with each layer adding another layer of abstraction before starting the next layer on top. For example, on Apple's iOS devices, the Boot ROM checks the signature on their Low-Level Bootloader before it allows it to load. When the Low-Level Bootloader finishes its tasks, it verifies and runs the next-stage bootloader, iBoot, which in turn verifies and runs the iOS kernel (*iOS Security* 2014). Whilst this is important work, we believe that ensuring the correct code is running is a largely orthogonal problem to ensuring that this startup code is in fact correct.

In this thesis we examine the question, "how can we be confident that our startup code correctly initialises our system in conformance with a given initial configuration?". This is especially important in secure systems such as capability-based systems as the security of such a system generally requires the access control primitives to be correctly provisioned. For example, the authority confinement, integrity, and non-interference proofs for the seL4 microkernel all rely on the capabilities of the system being configured in conformance with the access control policy (Sewell et al., 2011; Murray et al., 2013). Without this, these security results are null and void.

As an example, let us consider the initialisation and configuration of the example system from Section 3.4, reshown in Figure 5.1. To initialise this system, we first need to create these components, and then we configure each component by distributing the correct access rights to them, as well as setting up any other required data, such as stack pointers and instruction pointers. Only once these have been set up can we reason about the authority and information that can flow between the threads in our system. To have confidence that we have set up our system correctly, we need to know that the final initialised state is in conformance with the desired access control policy. We need to repeat this work for each system we wish to initialise. These systems we wish to initialise can be large. Andronick et al. (2010) analysed the security of a secure access controller, which, whilst a relatively simple system, had thousands of capabilities and thousands of objects to initialise, and ran Linux as one of the components. Clearly, this approach of specialised, hard-coded initialisers is not ideal, not only because of the tedious nature of initialising these systems, but also because we need to have confidence that we have done so correctly.

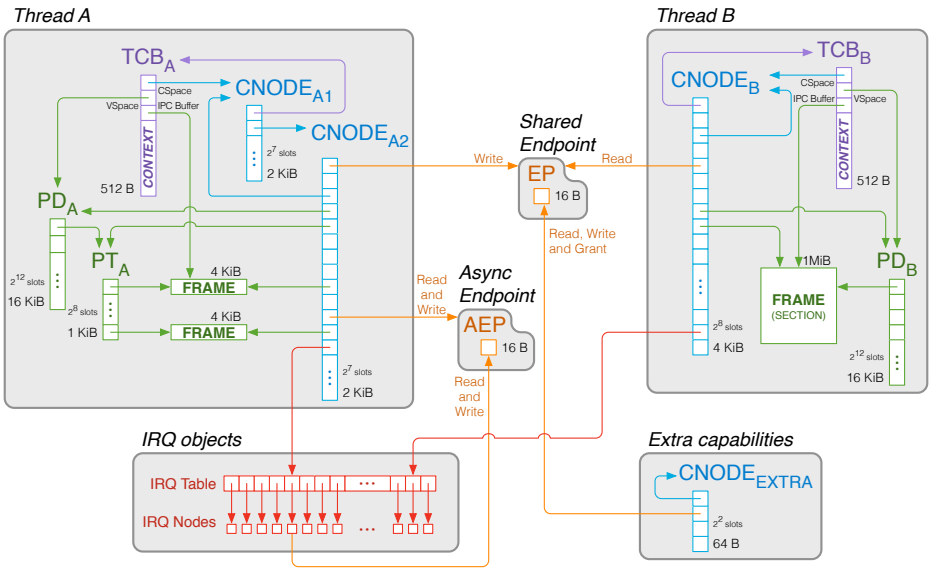


Figure 5.1: An example seL4 system, reshown from Figure 3.1.

Because of this, we think that automating and formally analysing the correct configuration of a system in conformance with a specified access control policy to be important work. Much of the previous work in this area has been at best formal with respect to high-level models of the operating systems themselves, without any connection to the operating system code itself.

In this thesis, we aim to extend the previous literature by providing a formal model of a general purpose algorithm for system initialisation on the seL4 microkernel. This lets us eliminate the tedious process of creating a different initialiser

for each different system we wish to initialise, whilst also allowing us to prove that this generic initialiser is correct *once*.

We build on a formal specification of the API of seL4 (explained in Section 7.4), which is written on a formal model of seL4 (the capDL kernel model, introduced in Section 3.4), which itself has been shown to be a formal abstraction of the seL4 code and formally linked to the seL4 binary. Connecting the proof of this algorithm to these seL4 proofs gives us great confidence that our algorithm is in fact correct.[1] The proof of this algorithm is described in detail in Chapter 7.

We specify the protection state of the desired system using capDL (as introduced in Section 3.4), which has been formally linked to the access control policies of Sewell et al. (2011) by Boyton et al. (2013). This gives us assurance that we can start systems in conformance with a security policy and know that the kernel will enforce these policies.

In the remainder of this chapter, we introduce how systems are initialised on the seL4 microkernel in Section 5.2, present a *formal algorithm* for system initialisation in Section 5.3, and present a *C implementation* of this algorithm in Section 5.4.[2] The relation between the latter two is illustrated in Figure 5.2. Finally, we conclude the chapter in Section 5.5.

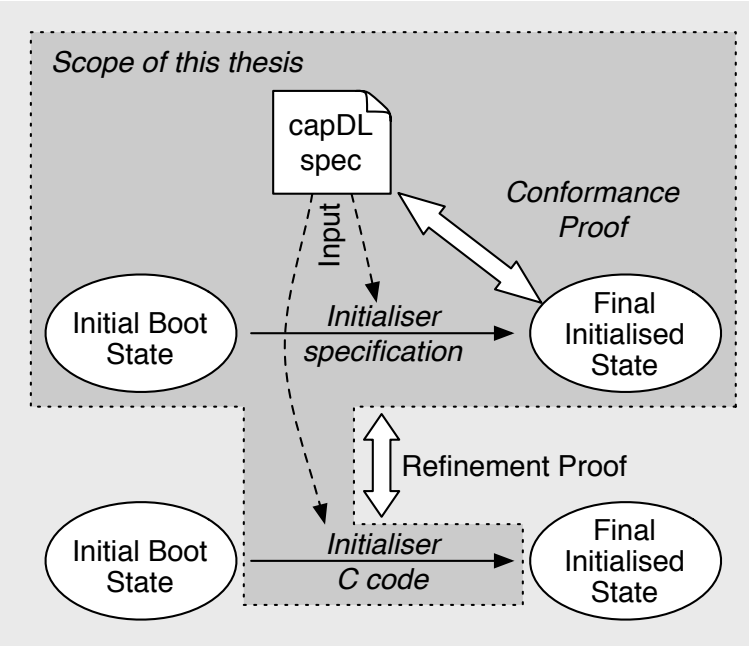


Figure 5.2: Overview of initialiser correctness proof.

[1] We only prove safety of our algorithm, not liveness, which reduces our confidence. We can demonstrate liveness through testing.

[2] The C implementation was done in conjunction with others.

5.2 seL4 initialisation

At boot time, seL4 first pre-allocates memory for itself and then gives the remainder of the memory to the initial user task (the *root task*) in the form of capabilities to untyped memory. More precisely, the kernel creates all the objects needed by the root task such as its TCB, capability space, and virtual address space. Capabilities to untyped memory are stored in the capability space, together with capabilities to allow hardware access. At the end of this booting phase, the root task is enabled to run and starts executing. The state of the system at this stage is illustrated in Figure 5.3.

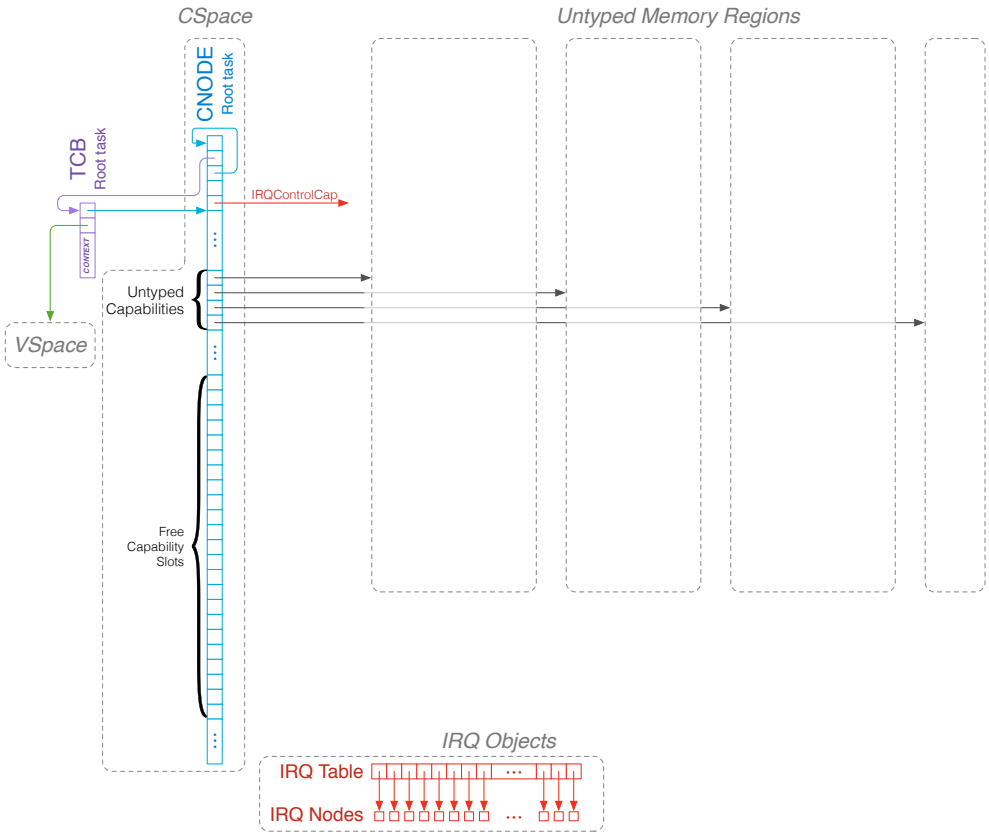


Figure 5.3: Initial state, after kernel booting.

This root task is the initialiser we are targeting in this thesis. Its aim is first to use these untyped capabilities for creating the required objects, such as TCB_A, TCB_B, and $\mathsf{CNode}_{\mathsf{A}_1}$ from Figure 5.1, and then to initialise them appropriately, for example, to set TCB_A's CSpace to $\mathsf{CNode}_{\mathsf{A}_1}$. This includes setting up communication channels, for example, storing the Write capability to EP in TCB_A's CSpace.

We present a *formal algorithm* for a custom root task to perform system initialisation. The initialiser specification describes the code for allocating objects, managing capabilities to the objects, copying or transferring authority, managing

```
example_spec =
(| cdl_arch = ARM11,
 cdl_objects =
        [tcb_a_id ↦      Tcb (| cdl_tcb_caps =
                                [ 0 ↦CNodeCap cnode_a1_id guard guard_sz cnode_a1_sz,
                                  1 ↦PageDirectoryCap pd_a_id Real None, …]
                                cdl_tcb_fault_endpoint = …,
                                cdl_tcb_intent = … |),
         tcb_b_id ↦      Tcb (| cdl_tcb_caps =
                                [ 0 ↦CNodeCap cnode_b_id guard guard_sz cnode_a1_sz,
                                  1 ↦PageDirectoryCap pd_b_id Real None, …]
                                cdl_tcb_fault_endpoint = …,
                                cdl_tcb_intent = … |),
    cnode_a1_id ↦ CNode (| cdl_cnode_caps =
                                [ 0 ↦TcbCap tcb_a_id,
                                  1 ↦CNodeCap cnode_a2_id guard guard_sz cnode_a2_sz, …]
    cnode_a2_id ↦ CNode (| cdl_cnode_caps =
                                [ 0 ↦EndPointCap ep_id 0 {Read, Write},
                                  1 ↦NullCap,
                                  2 ↦CNodeCap cnode_a2_id guard guard_sz cnode_a2_sz, …]
     cnode_b_id ↦ CNode …
 cnode_extra_id ↦ CNode …
          pd_a_id ↦ PageDirectory …]
…
 current_thread = … |)
```

Figure 5.4: capDL specification of Figure 5.1.

and mapping frames, and setting any required data (such as thread instruction pointers). Manually performing these tasks for each given system is complicated, error prone, and inflexible. Instead we choose to provide a *generic tool* that takes any capDL specification as input and *automatically* produces code to create and initialise the objects. Our initialiser takes a formal capDL specification describing the target system and runs to give a state where all the components have been created and their communication channels set up.

5.3 Formal model of system initialisation

In this section, we present an overview of our formal model of the system initialisation algorithm in Isabelle/HOL, and examine in detail the creation of objects, the creation of IRQ Handler capabilities, and the initialisation of the capability spaces as representative examples.

The system initialiser, being the first user task to run after boot time and having access to all available memory, is designed to take us from a boot state illustrated in Figure 5.3 into a state which is in conformance with a given capDL specification, like the one illustrated in Figure 5.1. We model the algorithm of the system initialiser as a sequence of high-level instructions, taking a formalised

capDL specification *spec* as input, and creating and initialising all objects and capabilities as specified by *spec*.

These formalised capDL specifications are produced by the capDL translator we introduced in Section 3.4. If we take the textual description of our example specification (see Figure 3.6) and run it through the capDL translator, we produce an Isabelle/HOL representation of the state equivalent[3] to the one illustrated in Figure 5.4.

Formally, *spec* has the type *cdl_state*, that is, a full state of the capDL kernel model. Its most important component is the kernel heap cdl_objects of type obj_id ⇒ cdl_object. CapDL objects are formalisations of the TCBs, CNodes, Endpoints, and other objects mentioned in Section 3.4. These objects consist of a map from capability slots to capabilities and potentially additional payload such as further TCB data.

The top-level definition of system initialisation init_system, shown in Figure 5.5, takes three parameters: the capDL specification *spec* to be initialised, the boot information *bootinfo* provided by the kernel to the initial user task which specifies the location of untyped memory and free capability slots in the initialiser's CSpace, and the list of object names *obj_ids* mentioned in *spec*. The system initialiser is purposely divided into well-defined separate phases, which simplifies reasoning as we will see in Chapter 7.

- Firstly, the initialiser processes the boot information provided by the kernel (first line of init_system body).
- Secondly, it
  - *a*) creates all the objects specified in the capDL specification while tracking the locations of the capabilities to these newly created objects and
  - *b*) creates all the required IRQ Handler capabilities while tracking the locations of these capabilities (second and third lines of init_system body).
- Thirdly, it duplicates some of the capabilities to the newly created objects in order to be able to later move some of those capabilities into some other component's CSpace, while keeping a copy of the capability for the initialiser to use itself (fourth line of init_system body).

[3] The precise formatting of the generated Isabelle/HOL specification is subtlety different to the definition we show here, with different parts of the definition folded into various constants. We can prove equivalence between the two definitions.

```
init_system spec bootinfo obj_ids ≡
do (ut_cpts, free_cptrs) ← parse_bootinfo bootinfo;
   (orig_caps, free_cptrs) ← create_objects spec obj_ids ut_cpts free_cptrs;
   (irq_caps, free_cptrs) ← create_irq_caps spec free_cptrs;
   dup_caps ← duplicate_caps spec orig_caps obj_ids free_cptrs;
   init_irqs spec orig_caps irq_caps;
   init_pd_asids spec orig_caps obj_ids;
   init_vspace spec orig_caps obj_ids;
   init_tcbs spec orig_caps obj_ids;
   init_cspace spec orig_caps dup_caps irq_caps obj_ids;
   start_threads spec dup_caps obj_ids
od
```

Figure 5.5: The top-level definition of the system initialiser model.

- Fourthly, it initialises each of these objects by type, including installing the capabilities into the capability storage objects (fifth to ninth lines of init_system body).

- Finally, it sets all threads to be runnable (tenth line of init_system body).

Parsing boot information

In the first phase of the initialiser, we extract from *bootinfo* the list *ut_cptrs* of capability pointers to the untyped memory regions that the initialiser has access to and can use to create new objects, as well as the list *free_cptrs* of capability pointers to free slots in its CSpace that it can use to store capabilities to these new objects. These lists of capabilities will each be in contiguous slots in the root task's capability space, as illustrated in Figure 5.3. The number of free capability slots provided to the root task is specified at compile time for seL4 and so it is possible to make sure that there will be enough free slots available for the initialisation of a specific system.

Creating objects and IRQ Handler capabilities

In the second phase, we create all objects and original capabilities listed in the capDL specification.[4]

Most capabilities in seL4, such as those pointing to Endpoints, TCBs, Page Tables, etc., point to exactly one object. The original capabilities pointing to these kinds of objects are created when the objects themselves are created (by retyping untyped memory). Such capabilities can be thought of as pointing to "real" objects.

[4]We consider IRQ Handler capabilities as original capabilities as they are not copied from other capabilities.

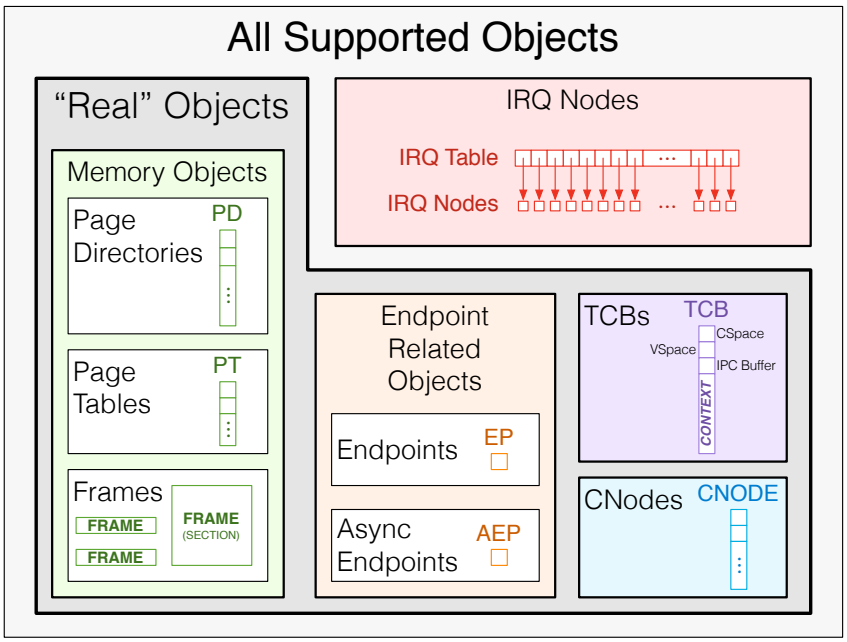


Figure 5.6: The different object types the initialiser supports.

The objects that the initialiser supports are illustrated in Figure 5.6.[5] Further copies of capabilities pointing to "real" objects can be derived from the original capability created when creating the object.

Other capabilities in seL4 do not point to an object at all. For example, IRQ Handler capabilities do not point directly to an object, but instead point to an IRQ number. This IRQ number can then be looked up in the IRQ table to work out which object the IRQ Handler capability ultimately points to. The IRQ Nodes in a system are created by kernel initialisation before the system initialiser starts. Because of this, to create IRQ Handler capabilities we do not create IRQ Nodes from untyped memory, but we instead invoke the seL4 system call seL4_IRQControl_Get. This system call requires the caller to possess an IRQ Control capability in its capability space. This IRQ Control capability is another example of a capability that does not point to an object, but confers the authority to create IRQ Handler capabilities.

Our formal algorithm of the system initialiser first creates the "real" objects and then creates the required IRQ Handler capabilities. The "real" objects are created by invoking seL4_Untyped_Retype on the provided untyped memory, as illustrated in Figure 5.7. Each untyped memory region can be retyped incrementally to create new objects. Any ordering of the creation of objects, and any choice of the untyped memory region they are created from, is safe. Some orderings may

[5] In the capDL kernel model, IRQ Nodes are represented as a CNode of size zero. As actual CNodes are never of size zero, we use the predicate real_cnode_at *obj_id spec* to specify that an actual CNode is present at *obj_id*.

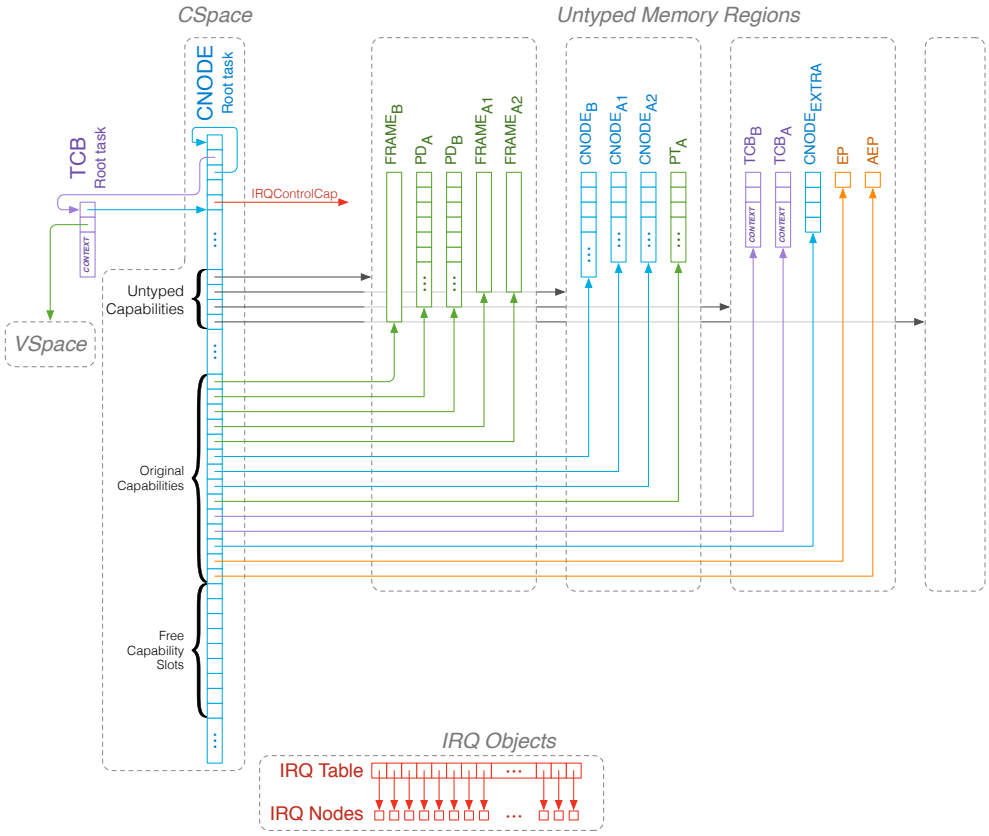


Figure 5.7: Kernel state, after create_objects.

however, make inefficient use of space as seL4 requires all objects to be aligned to their size. Ordering the creation of objects from the largest object to the smallest, and from the largest untyped memory region to the smallest, is conjectured to be optimal, but proving this is outside the scope of this work. In our algorithm, the creation is done in order of appearance in the list *obj_ids*.[6]

The operation create_objects, shown in Definition 5.1, has a large loop that creates each of the objects in the specification in the free slots of the initialiser's CSpace. We have three lists which we iterate over, the list of object identifiers for "real" objects *real_obj_ids*, the list of free capability pointers *free_cptrs* where the capabilities to the newly created objects will be stored, and the list of capability pointers to the untyped objects *untyped_cptrs* from which the objects will be created. Each list has a corresponding index into it. In each iteration of the loop, we call retype_untyped to try and create the specified object, in the specified free slot, using the specified untyped capability.

If the retype_untyped operation succeeds, then we will have created our specified kernel object, and have an original capability pointing to it in the free capa-

[6] In our implementation, the ordering is provided by the capDL translator.

bility slot that was provided to retype_untyped. In this case we then store in our mapping *orig_caps* the mapping stating that the original capability for the kernel object corresponding to the capDL identifier *obj_id* is located in the capability slot *free_slot*. We then move on to the next object to create, and the next free capability slot.

If the retype_untyped operation fails, we assume the untyped object has no more free space to create more objects, and instead move to the next untyped object. The loop continues to execute while none of the indexes overflow. If they do overflow, then we have either created all of the required objects, or we have run out of either free slots or untyped objects. We can ensure that we do not run out of free slots by configuring seL4 to ensure that the root task has enough free slots. If we run out of memory, we abort initialisation through our final assertion $untyped\_index \neq |untyped\_cptrs|$.

Definition 5.1 ► create_objects

```
create_objects spec obj_ids ut_cptrs free_cptrs ≡
do real_obj_ids ← return [obj_id←obj_ids. real_object_at obj_id spec];
   (obj_id_index, ut_index, orig_caps) ←
   whileLoop (λ(obj_id_index, ut_index, orig_caps).
                 (obj_id_index < |real_obj_ids| ∧ obj_id_index < |free_cptrs| ∧ ut_index < |ut_cptrs|,
                  do obj_id ← return real_obj_ids[obj_id_index];
                     free_cptr ← return free_cptrs[obj_id_index];
                     ut_cptr ← return ut_cptrs[ut_index];
                     object ← assert_opt $ opt_object obj_id spec;
                     object_type ← return $ object_type object;
                     object_size ← return $ object_size object;
                     fail ← retype_untyped free_cptr ut_cptr object_type object_size;
                     obj_id_index ← inc_when (¬ fail) obj_id_index;
                     ut_index ← inc_when fail ut_index;
                     orig_caps ← update_when (¬ fail) orig_caps obj_id free_cptr;
                     return (obj_id_index, ut_index, orig_caps)
                  od))
    (0, 0, map_empty);
   assert (ut_index ≠ |ut_cptrs|);
   return (orig_caps, drop |real_obj_ids| free_cptrs)
od
```

The operation retype_untyped, shown in Definition 5.2, is a simple wrapper around the seL4 system call seL4_Untyped_Retype, passing the required parameters to create a single object from the untyped capability pointed to by the capability pointer *ut_cptr*, placing the newly created capability in the capability slot in the root task's CNode that is pointed to by the capability pointer *free_cptr*.

◀ **Definition 5.2**
retype_untyped

```
retype_untyped free_cptr ut_cptr type size_bits ≡
seL4_Untyped_Retype ut_cptr type size_bits seL4_CapInitThreadCNode 0 0 free_cptr 1
```

To create the required IRQ Handler capabilities, we call create_irq_cap for each IRQ number used in the specification, that is, those that have an IRQ Handler capability pointing to them. This is shown in Definition 5.3. These capabilities are stored in the remaining free capability slots in order. The mapping of IRQ numbers to the capability pointer that points to the respective IRQ Handler capability is then calculated by zipping the list of IRQ numbers and free capability slots.

To create an IRQ Handler capability at the specified free capability slot, we call the seL4 system call seL4_IRQControl_Get, as shown in Definition 5.4. As mentioned above, this operation requires the initialiser to possess the IrqControlCap, which seL4 provides to the initialiser in the slot seL4_CapIRQControl, just as seL4 provides the capability to the root CNode in the slot seL4_CapInitThreadCNode.

◀ **Definition 5.3**
create_irq_caps

```
create_irq_caps spec free_cptrs ≡
do irqs ← return $ used_irqs spec;
   mapM (create_irq_cap spec) (zip irqs free_cptrs);
   si_irq_caps ← return $ map_of (zip irqs free_cptrs);
   return (si_irq_caps, drop |irqs| free_cptrs)
od
```

Once we have created all of the "real" objects and the IRQ Handler capabilities, the kernel state for our example will look like Figure 5.8.

◀ **Definition 5.4**
create_irq_cap

```
create_irq_cap spec (irq, free_cptr) ≡
do control_cap ← return seL4_CapIRQControl;
   root ← return seL4_CapInitThreadCNode;
   index ← return free_cptr;
   depth ← return 0x20;
   fail ← seL4_IRQControl_Get control_cap irq root index depth;
   assert (¬ fail)
od
```

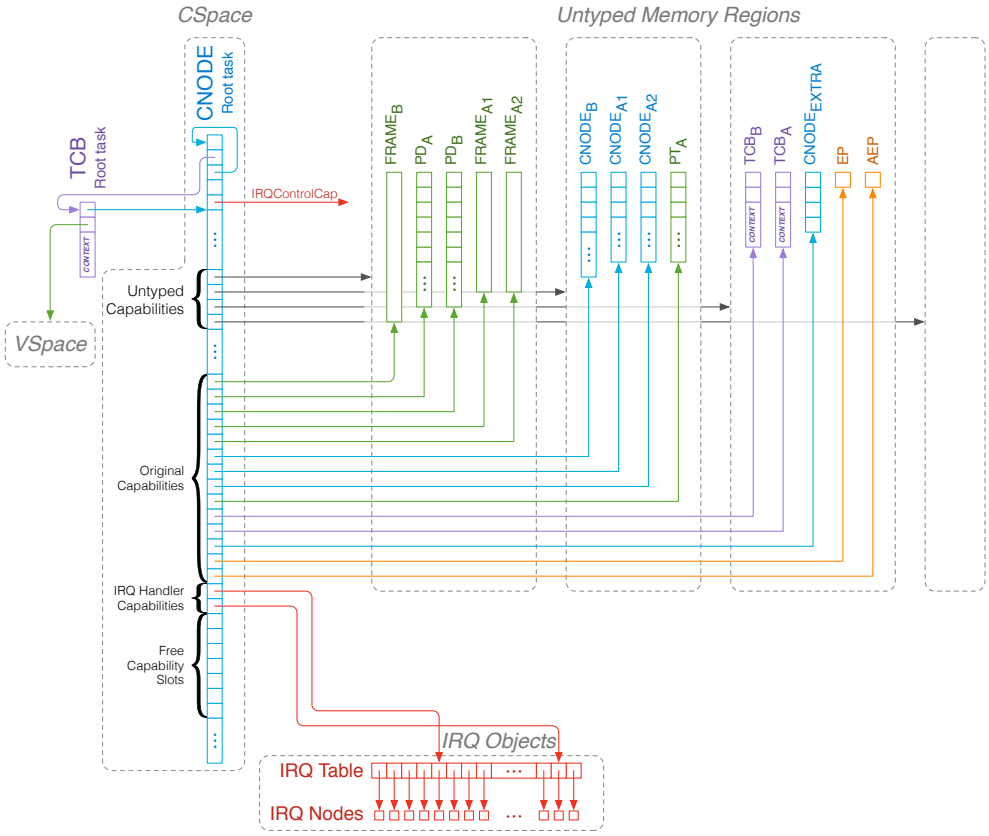


Figure 5.8: Kernel state, after create_irq_caps.

Duplicating capabilities

The original capability that is created when creating an object can then be given to other threads, either by moving it or by copying it (with full or diminished rights). Note that, as explained in Section 3.4, an original capability confers more authority than derived ones. This creates a subtle dependency for the order in which the initialiser has to distribute capabilities: it eventually needs to give away original capabilities, and at the same time keep access to the CNodes to finish their initialisation and the TCBs so it can start all the threads.

For this reason, we duplicate, in this third phase of system initialisation, the original capabilities to all the TCBs and CNodes created, storing these capabilities in the initialiser's CSpace, as illustrated in Figure 5.9. We store the mapping from object identifiers to the locations of these duplicated capabilities in *dup_caps*.

Configuring objects

At this stage we can start the initialisation, per object type, including installing the capabilities into the capability storage objects, as follows:

- IRQ Nodes are bound to asynchronous endpoints.

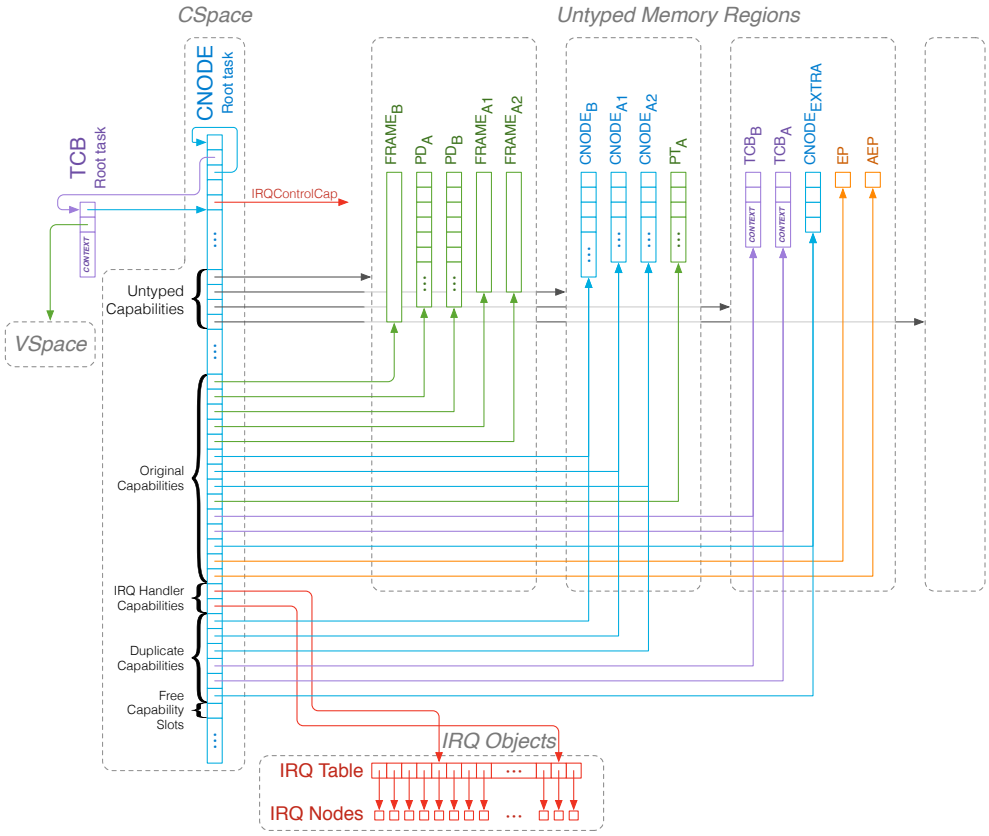


Figure 5.9: Kernel state, after duplicate_caps.

- Page directories each need to be assigned an address-space identifier (ASID)[7]. The ASIDs are all assigned from the system initialiser's ASID Pool.
- VSpaces are initialised by mapping in the required entries into page directories, and then page tables.
- TCBs are each initialised by setting the required data and capabilities, and then setting the registers.
- CSpaces are initialised similarly to VSpaces with the added complication of needing to distinguish between capabilities the that are *moved* and the ones that are *copied*. Moreover, unlike VSpaces which are fixed, two-level data structures, CSpaces can be arbitrary directed graphs.

The initialisation of CSpaces, shown in Definition 5.5, consists of putting the desired capability in every slot of every CNode appearing in *spec*. This occurs in two phases, depending on whether or not the *spec* requires the capability to be the original. For all capabilities that need not be originals, we *copy* the initialiser's

[7] seL4 requires all address spaces to be assigned a valid identifier.

original capability into the target CNode (we actually *mint* it, diminishing the access rights to those specified in *spec*). For the ones that need to be original, we *move* the initialiser's original capability (we actually *mutate* it with the appropriate rights specified in *spec*, except for endpoint capabilities which cannot be mutated in seL4). Each phase maps over the full list of all CNode slots, but does nothing to slots not affected by that phase.

Definition 5.5 ▶ init_cspace

```
init_cspace spec orig_caps dup_caps irq_caps obj_ids ≡
do cnode_ids ← return [obj_id←obj_ids. real_cnode_at obj_id spec];
   mapM (init_cnode spec orig_caps dup_caps irq_caps Copy) cnode_ids;
   mapM (init_cnode spec orig_caps dup_caps irq_caps Move) cnode_ids
od
```

In each phase, we initialise the capability slots one by one, as shown in Definition 5.6.

Definition 5.6 ▶ init_cnode

```
init_cnode spec orig_caps dup_caps irq_caps mode cnode_id ≡
do cnode_slots ← return $ slots_of_list spec cnode_id;
   mapM (init_cnode_slot spec orig_caps dup_caps irq_caps mode cnode_id) cnode_slots
od
```

The initialisation of a single capability slot *cnode_slot* of a CNode *cnode_id* is shown in Definition 5.7. This could, for instance, be the first slot of $\mathsf{CNode}_{\mathsf{A_1}}$ in our example of Figure 5.1, which needs to contain a non-original capability to the endpoint EP with a Write right.

In this definition, we first extract the target capability *target_cap* that *spec* requires in *cnode_slot*. The function opt_cap returns an *option* type, that is, either Some *cap* or None. The function assert_opt asserts that this value is of the form Some *cap* and returns *cap*; otherwise it fails. From *target_cap*, we extract the target object *target_cap_obj* (say EP), the desired rights *target_cap_rights* (say Write), and additional data *target_cap_data* (for example, for endpoints, a so-called *badge*). We store in move_cap whether *spec* requires the capability stored in the slot to be original (in which case, we move the capability).

◀ **Definition 5.7** init_cnode_slot

```
init_cnode_slot spec orig_caps dup_caps irq_caps mode cnode_id cnode_slot ≡
do target_cap ← assert_opt (opt_cap (cnode_id, cnode_slot) spec);
   target_cap_obj ← return (cap_object target_cap);
   target_cap_irq ← return (cap_irq target_cap);
   target_cap_rights ← return (cap_rights target_cap);
   target_cap_data ← return (cap_data target_cap);
   is_ep_cap ← return (ep_related_cap target_cap);
   is_irqhandler_cap ← return (is_irqhandler_cap target_cap);
   move_cap ← return (is_orig_cap (cnode_id, cnode_slot) spec);
   dest_obj ← assert_opt $ opt_object cnode_id spec;
   dest_size ← return (object_size dest_obj);
   dest_root ← assert_opt (dup_caps cnode_id);
   dest_index ← return cnode_slot;
   dest_depth ← return dest_size;
   src_root ← return seL4_CapInitThreadCNode;
   src_index ← assert_opt (if is_irqhandler_cap
                           then irq_caps target_cap_irq
                           else orig_caps target_cap_obj);
   src_depth ← return 0x20;
   if target_cap = NullCap then return True
   else if mode = Move ∧ move_cap
        then if is_ep_cap ∨ is_irqhandler_cap
             then seL4_CNode_Move dest_root dest_index dest_depth
                                  src_root src_index src_depth
             else seL4_CNode_Mutate dest_root dest_index dest_depth
                                    src_root src_index src_depth target_cap_data
        else if mode = Copy ∧ ¬ move_cap
             then seL4_CNode_Mint dest_root dest_index dest_depth
                                  src_root src_index src_depth
                                  target_cap_rights target_cap_data
             else return True
od
```

In order to be able to invoke seL4's seL4_CNode_Move, seL4_CNode_Mutate, and seL4_CNode_Mint operations, the initialiser needs to hold in its CSpace, both the target capability to be moved or copied, and a capability to the destination slot. We compute the destination information (*dest_root*, *dest_index*, and *dest_depth*) from the (duplicated) capability that the initialiser holds for the destination slot. (We use the duplicate capabilities in case the original capability to the destination slot has already been given away.) We compute the source information (*src_root*,

src_index, and *src_depth*) from the (original) capability that the initialiser holds for the target capability. We then can invoke the appropriate seL4 operation depending if the target capability needs to be original or not. These operations directly connect to the capDL-level API model of the kernel.

Once create_objects has finished, all the original and IRQ Handler capabilities will have been handed out, and the state will look like the one illustrated in Figure 5.10.

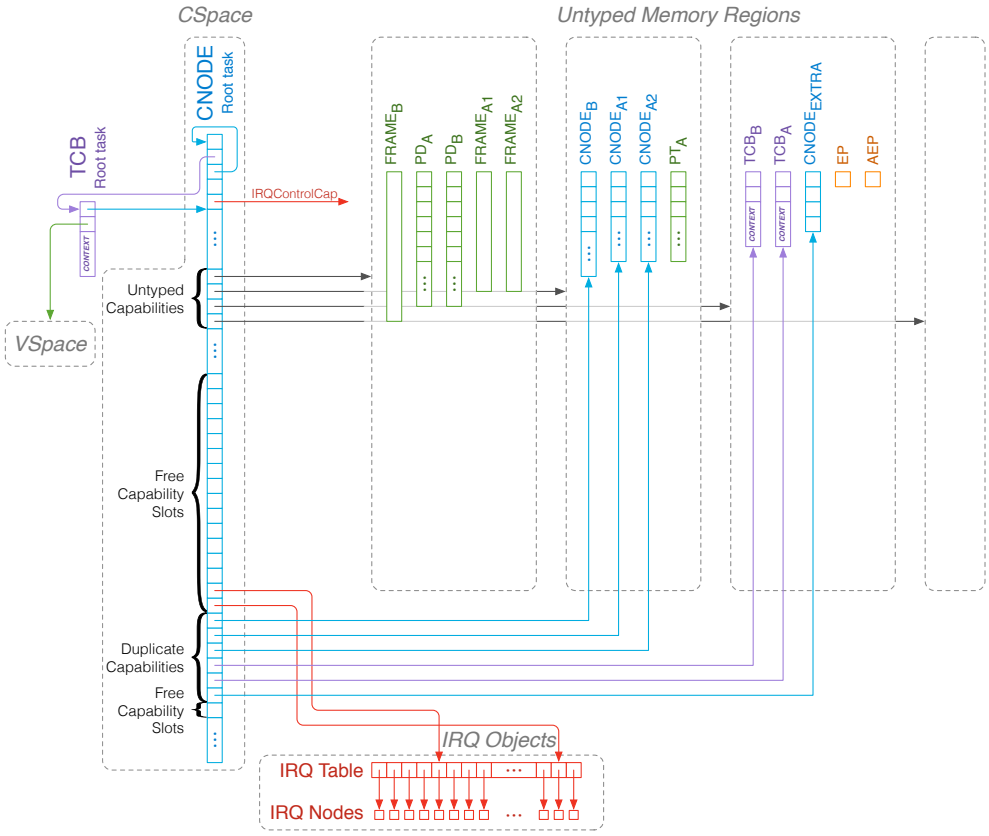


Figure 5.10: Kernel state, after init_cspace. Note, capabilities *between* objects being initialised are not shown for simplicity.

Starting threads

The final step of the initialisation is to set all threads to be runnable, at which point the initialiser becomes dormant and the system is ready to run. As the scheduling of threads is not modelled, we have not proven that this last step is correct.

5.4 C implementation

In this section, we take the formal algorithm of system initialisation from Section 5.3 and show how it has been implemented in C. This C implementation was done in conjunction with others. The implementation follows the formalised algorithm closely, such that a formal refinement between the two is possible, with functional aspects such as mapM and filter turned into their C equivalents. As a representative example we examine the initialisation of capability spaces in detail.

The top level-definition of the C function `init_system`, shown in Figure 5.11, closely mirrors the Isabelle/HOL definition of init_system.

```
void init_system(const CDL_Model *spec) {
    seL4_BootInfo *bootinfo = seL4_GetBootInfo();
    parse_bootinfo(bootinfo);

    create_objects(spec, bootinfo);
    create_irq_caps(spec);

    duplicate_caps(spec);

    init_irqs(spec);
    init_pd_asids(spec);
    init_vspace(spec);
    init_tcbs(spec);
    init_cspace(spec);
    start_threads(spec);
}
```

Figure 5.11: The C implementation of `init_system`.

The initialiser constructs a number of large data structures, like the mapping from object identifiers to the capability slot that the original capability to the corresponding kernel object is stored in—*orig_caps* in the Isabelle/HOL model. Rather than passing these data structures between functions and using large amounts of stack space, or dynamically allocating memory for them in the heap and passing pointers to them (which requires `malloc`, a feature best avoided in an initialiser), we store these large data structures in global variables. This helps us ensure that the initialiser will not run out of either stack or heap space at runtime.

The boot information provided by the kernel is obtained through the seL4 provided function `seL4_GetBootInfo()`. The boot information provided two things required by the initialiser, the list of free capability slots and the list of untyped capability pointers. As explained earlier, seL4 provides these capabilities as contiguous regions, and so the boot information encodes each of these lists with two

capability pointers, one to the first capability slot and one to last capability slot. We store the capability pointers to the first and last free slots in two global variables, the former of which is updated as these free slots are consumed. The list of untyped capabilities is only required for creating objects, and so this information is passed into the `create_objects` function by passing in the boot information data structure.

As we stated above, the C code has been written in such a way that we can prove refinement between the C code and the formal algorithm for system initialisation. To do such a refinement, we need to translate our C code into the theorem prover Isabelle/HOL. One promising new way to do this is to use the tool AutoCorres (Greenaway et al., 2012; Greenaway et al., 2014) which builds on an existing C-to-Isabelle parser (Norrish, 1998; Norrish, 2013). Using these tools requires our C code be written in a subset of C. This means that we cannot use C unions, function pointers, or take the address of local variables. As the seL4 system call `seL4_TCB_WriteRegisters()` takes a pointer to a C struct containing the values of the registers to be set, we need to store this structure in a global variable, updating its value before making each call to `seL4_TCB_WriteRegisters()`, passing the address of this global variable. As our code is single threaded, this is safe, although not particularly ideal.[8]

This gives us a total of six global variables in our program, three arrays encoding the mappings *orig_caps*, *dup_caps*, and *irq_caps* respectively, two global variables pointing to the start and end of the free capability slots of the initialiser, and one global variable to store the registers of a thread, as shown in Figure 5.12.

```
static seL4_CPtr capdl_to_sel4_orig[CONFIG_CAPDL_LOADER_MAX_OBJECTS];
static seL4_CPtr capdl_to_sel4_copy[CONFIG_CAPDL_LOADER_MAX_OBJECTS];
static seL4_CPtr capdl_to_sel4_irq[CONFIG_CAPDL_LOADER_MAX_OBJECTS];

static seL4_CPtr free_slot_start, free_slot_end;

static seL4_UserContext global_user_context;
```

Figure 5.12: The global variables used by the C implementation.

The capDL specification parsed by the system initialiser is stored in a C data structure generated by the capDL tools described in Section 3.4. This data structure contains an array of the objects to be created; each object stores its type, a (possibly empty) array of capabilities, the object's size, and any other information required about the object.

[8] Another alternative would have been to change the `seL4_TCB_WriteRegisters` system call to take the structure by value.

Since object identifiers used by capDL are simply identifiers, we assume in our implementation that object identifiers are continuous and start at zero.[9] Objects are initialised by the ordering of the array in which they are stored (meaning we do not need the list of objects that the Isabelle/HOL specification has).

To examine exactly how we map a monadic Isabelle/HOL function to a C implementation, we examine the definition of `init_cpsace` in detail, shown in Figure 5.13. In the Isabelle/HOL specification of the algorithm, init_cspace maps the operation init_cnode over the list of CNodes twice, once to copy capabilities, and once to move capabilities. The list of CNodes is obtained by filtering the list of objects. In C, we achieve this by looping over the list of objects, and calling `init_cnode` on each object that is a CNode. The mode (`MOVE` or `COPY`) is encoded using an enumeration `init_cnode_mode`.

```
void init_cspace(const CDL_Model *spec) {
    for (CDL_ObjID obj_id = 0; obj_id < spec->num; obj_id++)
        if (spec->objects[obj_id].type == CDL_CNode)
            init_cnode(spec, COPY, obj_id);

    for (CDL_ObjID obj_id = 0; obj_id < spec->num; obj_id++)
        if (spec->objects[obj_id].type == CDL_CNode)
            init_cnode(spec, MOVE, obj_id);
}
```

Figure 5.13: The C implementation of `init_cspace`.

To initialise a CNode, we initialise each of the capability slots within, as shown in Figure 5.14. The capability slots of an object can be a sparse data structure, and so we encode it as an array of `CDL_CapSlots`, which each contain a slot number and a capability. We index into this array with `slot_index`.

```
void init_cnode(const CDL_Model *spec, init_cnode_mode mode, CDL_ObjID cnode_id) {
    CDL_Object *cdl_cnode = get_spec_object(spec, cnode_id);
    for (unsigned int slot_index = 0; slot_index < CDL_Obj_NumSlots(cdl_cnode);
         slot_index++)
        init_cnode_slot(spec, mode, cnode_id, CDL_Obj_GetSlot(cdl_cnode,
             slot_index));
}
```

Figure 5.14: The C implementation of `init_cnode`.

[9] In fact, in our textual descriptions, object are identified by a string, and the capDL translator generates numerical identifiers for the objects, sorted from largest object to smallest to ensure allocation is optimal.

```
void init_cnode_slot(const CDL_Model *spec, init_cnode_mode mode,
                     CDL_ObjID cnode_id, CDL_CapSlot *cnode_slot) {
    CDL_Cap *target_cap = CDL_CapSlot_Cap(cnode_slot);
    CDL_ObjID target_cap_obj = CDL_Cap_ObjID(target_cap);
    CDL_IRQ target_cap_irq = CDL_Cap_IRQ(target_cap);
    seL4_CapRights target_cap_rights = CDL_Cap_Rights(target_cap);
    seL4_CapData_t target_cap_data = get_capData(CDL_Cap_Data(target_cap));
    CDL_CapType target_cap_type = CDL_Cap_Type(target_cap);

    int is_ep_cap          = (target_cap_type == CDL_EPCap ||
                              target_cap_type == CDL_AEPCap);
    int is_irq_handler_cap = (target_cap_type == CDL_IRQHandlerCap);
    int is_orig_cap        = CDL_Cap_IsOrig(target_cap);

    CDL_Object *dest_obj = get_spec_object(spec, cnode_id);
    uint8_t dest_size = CDL_Obj_SizeBits(dest_obj);

    seL4_CPtr dest_root = dup_caps(cnode_id);
    int dest_index      = CDL_CapSlot_Slot(cnode_slot);
    uint8_t dest_depth  = dest_size;

    seL4_CPtr src_root = seL4_CapInitThreadCNode;
    int src_index = is_irq_handler_cap ? irq_caps(target_cap_irq)
                                       : orig_caps(target_cap_obj);
    uint8_t src_depth = 32;

    if ((mode == MOVE) && is_orig_cap) {
        if (is_ep_cap || is_irq_handler_cap) {
            int error = seL4_CNode_Move(dest_root, dest_index, dest_depth,
                                        src_root, src_index, src_depth);
            seL4_AssertSuccess(error);
        } else {
            int error = seL4_CNode_Mutate(dest_root, dest_index, dest_depth,
                                          src_root, src_index, src_depth,
                                          target_cap_data);
            seL4_AssertSuccess(error);
        }
    } else if ((mode == COPY) && !is_orig_cap) {
        int error = seL4_CNode_Mint(dest_root, dest_index, dest_depth,
                                    src_root, src_index, src_depth,
                                    target_cap_rights, target_cap_data);
        seL4_AssertSuccess(error);
    }
}
```

Figure 5.15: The C implementation of `init_cnode_slot`.

To initialise a CNode slot, we read the required data from the capDL specification, as shown in Figure 5.15, and call either `seL4_CNode_Mutate`, `seL4_CNode_Move`, or `seL4_CNode_Mint`. Either way we assert that the seL4 system call succeeded.

5.5 Conclusion

In this chapter we have presented a formal algorithm of system initialisation that takes a declarative capDL specification and starts a system in conformance with this specification. At present, capDL only models the protection state of the system, not its memory content. This means we also do not model the loading of program code. This limitation is less severe than it may sound, because in the envisioned application space, the system image loaded from disk already contains all application binaries. That is, loading program code is reduced to mapping the right memory frames into the right virtual address spaces, which we do model.

This algorithm we have presented here creates and initialises the specified objects one-by-one. In the next chapter we introduce separation logic, a reasoning framework that allows us to conveniently reason that each step of this algorithm brings us one step closer to having an initialised system, whilst also proving that each step does not undo any of the work done by previous steps.

Chapter Summary

- Correctly setting up a MILS-style system to conform to a specification is important for the security of the system.
- To solve this problem, we presented a generic, formal algorithm that initialises a system in conformance with a declarative capDL specification. Our algorithm takes a capDL specification as input, and transforms the initial seL4 boot state to a final initialised state in conformance with our given specification.
- We presented a C implementation of this algorithm that has been designed such that we could prove that it refines our formal algorithm.

Chapter 6

Separation logic

In this chapter we introduce the logic that we use for the proof in this thesis, namely, separation logic.

- We start in Section 6.1 by introducing Hoare logic and the limitations of using Hoare logic for the proof of the algorithm for our system initialiser.
- We then introduce separation logic, an extension of Hoare logic first proposed by Reynolds (2002), in Section 6.2, and we explain why it is useful for reasoning about our system initialiser.
- In Section 6.3 we explain an abstract separation logic developed by Klein et al. (2012), and show how it can be used to avoid creating yet another separation logic from scratch.
- In Section 6.4, we show a number of properties true for all separation logics, concentrating mainly on some new properties that we show for this thesis.
- We introduce our custom separation logic instance in Section 6.5. We show how we can achieve a fine level of granularity with a simple instantiation of an abstract separation logic, and we explain why we have chosen the level of granularity that we have.
- In Section 6.6, we develop a number of predicates for describing the state—the so called "arrows" of a separation logic. These are what let us reason about whole objects, or components of these objects such as the fields or the individual capabilities.
- We show a variation of the classical frame rule of separation logic, on a shallow embedding, in Section 6.7. This is what enables us to show that an operation's behaviour is local.

- Finally we conclude this chapter in Section 6.8 with a summary of the results.

Parts of this chapter have previously appeared in Boyton et al. (2013) and Andronick et al. (2012).

6.1 Why not use standard Hoare logic?

When reasoning about the system initialiser specification, we wish to precisely capture the semantics of each operation. Axiomatic semantics are commonly represented in *Hoare logic*, using *Hoare triples*. These triples, written $\{P\}\ f\ \{Q\}$, express that if P is true before an operation f, then Q will be true afterwards.[1]

To understand what these semantics look like, we will examine the set_object operation, defined on a simple object heap, like the object heap in capDL. We model memory as a partial mapping from object identifiers to objects; at any memory address an object may or may not exist. In this world, we define the set_object operation as storing an object *object* at a memory address *obj_id*. In this section, we use the notation $[obj\_id] := \mathsf{Some}\ object$ to place the object *object* at the location *obj_id*.

Definition 6.1 ► set_object

$$\mathsf{set\_object}\ obj\_id\ object \equiv [obj\_id] := \mathsf{Some}\ object$$

The following Hoare triple describes the behaviour of the monadic operation set_object. This Hoare triple precisely captures the fact that given any precondition, if we run the operation set_object, then afterwards *obj_id* will point to *object* as desired. The notation "$obj\_id \hookrightarrow object\ s$" means that there exists an object *object* at a memory address *obj_id* in the state s.

Lemma 6.1 ► set_object *sets object*

$$\begin{array}{l}\{\lambda s.\ \mathsf{True}\}\\ \quad \mathsf{set\_object}\ obj\_id\ object\\ \{\lambda s.\ obj\_id \hookrightarrow object\ s\}\end{array}$$

To be able to chain multiple executions of this rule, we need to know two things about set_object, namely:

1. that it is local
2. that the objects that set_object modifies are distinct.

[1] In this thesis, we only reason about partial correctness—if f terminates, then Q will be true. Termination could be proven separately, if desired.

Showing that an operation is local means both that its behaviour is unaffected by the state of the rest of the system, and that it does not affect the rest of the system; it lets us be explicit about what the side-effects of an operation are. Our Hoare triple, shown in Lemma 6.1, shows that the object located at the address *obj_id*, after running set_object *obj_id object*, will be *object*, independent of the rest of the state, but it does not show anything about how set_object could potentially modify the contents of the rest of the state. To illustrate this point, let us examine set_object_malicious, which not only sets the object at location *obj_id* to *object*, but also deletes the next object (the one located at $obj\_id + 1$).[2]

◀ **Definition 6.2** set_object_malicious

```
set_object_malicious obj_id object ≡
do [obj_id] := Some object;
   [obj_id + 1] := None
od
```

Now, we can show the same Hoare triple that we showed for set_object, namely Lemma 6.1, is also true for set_object_malicious.

◀ **Lemma 6.2** set_object_malicious *sets object*

$$\{\lambda s.\ \mathsf{True}\}$$
$$\quad \mathsf{set\_object\_malicious}\ obj\_id\ object$$
$$\{\lambda s.\ obj\_id \hookrightarrow object\ s\}$$

As we can see, our Hoare triple in Lemma 6.1 is not nearly precise enough to capture the locality of set_object. To fix this, we can show another Hoare triple for set_object that states that the rest of the objects in the system are unchanged.

◀ **Lemma 6.3** set_object *leaves other objects*

$$\{\lambda s.\ obj\_id' \hookrightarrow object'\ s \wedge obj\_id \neq obj\_id'\}$$
$$\quad \mathsf{set\_object}\ obj\_id\ object$$
$$\{\lambda s.\ obj\_id' \hookrightarrow object'\ s\}$$

Lemma 6.3 states that any other objects in the system are unchanged by the setting of *obj_id* to point to *object*. Combining this lemma with Lemma 6.1, we can capture exactly what set_object does and what it does *not* do. This allows us say that set_object is *local*.

We can then chain Lemma 6.1 and Lemma 6.3 together to show that the setting of two objects, at distinct addresses, sets them in the expected manner.

[2] Alternatively, we could consider a version of set_object_malicious that could exhibit different behaviour depending on the contents of the state, as long as it always places *object* at *obj_id*.

Lemma 6.4 ► *Setting two objects*

$$\{\lambda s.\ obj\_id \neq obj\_id'\}$$
$$\text{do set\_object } obj\_id\ object;$$
$$\quad \text{set\_object } obj\_id'\ object'$$
$$\text{od}$$
$$\{\lambda s.\ obj\_id \hookrightarrow object\ s \wedge obj\_id' \hookrightarrow object'\ s\}$$

Proof: Because we know the *obj_id* and *obj_id′* are different, we know that the two calls to set_object will affect different objects. We chain Lemma 6.1 and Lemma 6.3 (which together show that set_object is a local operation) using the weakest-precondition framework of Cock et al. (2008) prove this lemma. ■

This style of reasoning applies in general—to chain rules we need to know that they are local and that their areas of influence do not overlap. Unfortunately, while this style of reasoning scales to atomically setting two objects, it fails to scale well to setting parts of an object (such as an individual capability slot), or reasoning about operations that require access to multiple different objects, as specifying and reasoning about the locality and disjointness of these operations becomes increasingly unwieldy.

6.2 A simple introduction to separation logic

Separation logic—an extension to Hoare logic first introduced by Reynolds (2002)—allows us to express these two concepts of *locality* and *disjointness* implicitly, by local reasoning on *partial heaps*. These partial heaps, just like the object heap in capDL, are partial mappings from object identifiers to objects.

Using a separation logic allows us to express the notion that the fact that *obj_id* points to *object* and that *obj_id′* points to *object′* are both true, but in disjoint parts of the heap. To do this, we construct separation logic predicates, each operating on a piece of the heap, stating precisely which object is present and that only this object is present.[3] In our example heap, we define such a predicate as follows:

Definition 6.3 ► sep_map

$$obj\_id \mapsto_{\text{obj}} object \equiv \lambda s.\ \text{cdl\_objects}\ s = [obj\_id \mapsto object]$$

[3] Separation logics do not require that predicates specify that the objects specified are the *only* objects in the heap, but predicates that do are much easier to reason about using weakest-precondition style reasoning. For this reason, all the separation logic predicates used in this thesis constrain the heap to only contain the specified objects.

This predicate states that the heap contains *only* the object *object* located at address *obj_id*. We define another similar predicate that states that there exists *any* object at an address *obj_id*.

$$obj\_id \mapsto_{\mathsf{obj}} - \equiv \lambda s.\ \exists object.\ \mathsf{cdl\_objects}\ s = [obj\_id \mapsto object]$$

◀ **Definition 6.4**
sep_any_map

We can use these separation logic predicates to restate a more precise form of our original rule about set_object. To do so, we divide our object heap in two, and say that the first piece of the heap previously contained some object located at address *obj_id*, and that after set_object is called, that piece of the heap contains the object *object* located at address *obj_id*.

$$\begin{array}{l}\{obj\_id \mapsto_{\mathsf{obj}} - \wedge^* R\} \\ \ \mathsf{set\_object}\ obj\_id\ object \\ \{obj\_id \mapsto_{\mathsf{obj}} object \wedge^* R\}\end{array}$$

◀ **Lemma 6.5**
Set object

What is different about this rule is that it states that the rest of the object heap, as captured by the arbitrary separation logic predicate *R*, is unchanged. Since this Hoare triple is true for *any R*, the behaviour of set_object can neither depend on any of the rest of the state, nor can it modify the rest of the state. This result is achieved using separation conjunction $P \wedge^* Q$. Separation conjunction is what allows us to say that a predicate *P* and predicate *Q* are true on *separate* parts of the state.

To define separation conjunction, we need to formalise what it means to divide a heap into two pieces. To do this we, require a concept of what it means for two heaps, h_o and h_1, to be disjoint ($h_o \#\# h_1$), and how we can combine disjoint heaps together ($h_o + h_1$).[4] Using these concepts of heap disjunction and heap addition, we define separation conjunction.

$$P \wedge^* Q \equiv \lambda h.\ \exists h_o\ h_1.\ h_o \#\# h_1 \wedge h = h_o + h_1 \wedge P\ h_o \wedge Q\ h_1$$

◀ **Definition 6.5**
Separation conjunction

We say that the predicate *P* is true, separately to the predicate *Q*, when we can divide our heap *h* into two disjoint parts, h_o and h_1, such that $h = h_o + h_1$, where *P* is true on h_o, and *Q* is true on h_1. For simple object heaps, such as the one in Section 6.1, we can define heap addition as map addition between the two object

[4] In a classical separation logic, heap addition is a partial function, and heaps are considered disjoint whenever heap addition is defined. As reasoning about partial functions is inconvenient in Isabelle/HOL, we follow Klein et al. (2012) and use total functions for both heap addition and heap disjunction.

heaps, and heap disjunction as map disjunction, to construct a separation logic with an object-level granularity (a concept we explore further in the rest of this chapter).

This simple separation logic that we have introduced in this section, with its objet-level granularity and precise, local predicates, such as $obj\_id \mapsto_{\mathsf{obj}} object$, allow us to use separation conjunction to specify a rule for set_object that implicitly states both the locality and distinctness requirements from Section 6.1. We reason about the setting of two objects at distinct addresses simply by instantiating the R in Lemma 6.5 to include the other object.

Examining the same setting of two objects that we analysed in Lemma 6.4, that is, setting of obj_id to $object$ and obj_id' to $object'$, we use the rule Lemma 6.5 twice. The first time setting R to be $obj\_id' \mapsto_{\mathsf{obj}} - \wedge^* R$ and the second setting R to be $obj\_id \mapsto_{\mathsf{obj}} object \wedge^* R$, as shown below.

$$\{obj\_id \mapsto_{\mathsf{obj}} - \wedge^* obj\_id' \mapsto_{\mathsf{obj}} - \wedge^* R\}$$
$$\text{set\_object } obj\_id\ object$$
$$\{obj\_id \mapsto_{\mathsf{obj}} object \wedge^* obj\_id' \mapsto_{\mathsf{obj}} - \wedge^* R\}$$

$$\{obj\_id' \mapsto_{\mathsf{obj}} - \wedge^* obj\_id \mapsto_{\mathsf{obj}} object \wedge^* R\}$$
$$\text{set\_object } obj\_id'\ object'$$
$$\{obj\_id' \mapsto_{\mathsf{obj}} object' \wedge^* obj\_id \mapsto_{\mathsf{obj}} object \wedge^* R\}$$

We can chain these rules together, noting that separation conjunction is commutative.

Lemma 6.6 ▶ *Setting two objects*

$$\{obj\_id \mapsto_{\mathsf{obj}} - \wedge^* obj\_id' \mapsto_{\mathsf{obj}} - \wedge^* R\}$$
$$\text{set\_object } obj\_id\ object$$
$$\{obj\_id \mapsto_{\mathsf{obj}} object \wedge^* obj\_id' \mapsto_{\mathsf{obj}} - \wedge^* R\}$$
$$\text{set\_object } obj\_id'\ object'$$
$$\{obj\_id \mapsto_{\mathsf{obj}} object \wedge^* obj\_id' \mapsto_{\mathsf{obj}} object' \wedge^* R\}$$

Whilst this may not look much simpler than our similar proof of Lemma 6.4, it scales more easily to reasoning about loops, multiple objects, and, with a separation logic of the right granularity (not this simple one), parts of an object. The calculation for the value for R to use when applying a rule is called *frame calculation* and can be automated through the use of custom Isabelle/HOL tactics, as explained in Section 6.3. We explain the *frame* in our rules in further detail in Section 6.7.

6.3 An abstract separation logic

The definitions given in Section 6.2 give us a basic separation logic with the desired properties for the atomic setting of objects. Underneath a separation logic

is an algebraic structure which has been analysed by Calcagno et al. (2007), Dang et al. (2011), and Dockins et al. (2009). To be able to reason about a separation logic in a theorem prover such as Isabelle/HOL, custom tactics (such as those that do *frame calculation*) are highly useful in automating much of the proof. Rather than re-deriving the properties we need for our specific separation logic, such as the associativity and commutativity of separation conjunction, and reimplementing tactics for reasoning about them, we build the separation logic we use for this thesis on an abstract separation logic developed by Klein et al. (2012). This abstract separation logic formalises the work of Calcagno et al. (2007) in the theorem prover Isabelle/HOL and provides an Isabelle/HOL locale. This locale formalises the axioms of a separation algebra, and derives a number of properties, definitions, and tactics based on those axioms. To use this abstract separation logic, we are required to instantiate this locale with our own custom definitions of heap disjunction and heap addition, and then to prove that these custom definitions satisfy the axioms of the locale.[5] We then get a separation logic, along with a number of properties, definitions, and tactics. On top of this separation logic, we define predicates such as $obj\_id \mapsto_{\mathsf{obj}} object$, and prove the *frame rule* for our leaf functions.

$$h \,\#\#\, 0$$

◀ **Axiom 1** *Disjoint identity*

$$h_0 \,\#\#\, h_1 \Longrightarrow h_1 \,\#\#\, h_0$$

◀ **Axiom 2** *Disjoint commutativity*

$$h + 0 = h$$

◀ **Axiom 3** *Addition identity*

$$\frac{h_0 \,\#\#\, h_1}{h_0 + h_1 = h_1 + h_0}$$

◀ **Axiom 4** *Addition commutativity*

$$\frac{h_0 \,\#\#\, h_1 \qquad h_1 \,\#\#\, h_2 \qquad h_0 \,\#\#\, h_2}{(h_0 + h_1) + h_2 = h_0 + (h_1 + h_2)}$$

◀ **Axiom 5** *Addition associativity*

$$\frac{h_0 \,\#\#\, h_1}{h \,\#\#\, h_0 + h_1 = (h \,\#\#\, h_0 \wedge h \,\#\#\, h_1)}$$

◀ **Axiom 6** *Disjunction over addition*

To understand each of these axioms, let us examine them on a standard object heap, such as the object heap of capDL, and examine these axioms on the operations of map addition and map disjunction.[6] The intuition behind these axioms

[5] We instantiate a stronger version of the abstract separation logic of Klein et al. (2012) as it is slightly easier and more intuitive.

[6] As noted earlier, such an instantiation gives us a separation logic with object-level granularity. The separation logic we use for this thesis is more complicated, and explained in Section 6.5.

is that whenever heaps are disjoint, then heap addition should obey the standard axioms of addition.

As can be seen from the axioms, our algebra requires a zero. In our simple example of maps as heaps, the empty map is the zero of this algebra.

Definition 6.6 ► map_empty

$$\text{map\_empty} \equiv \lambda x.\ \text{None}$$

We define heap disjunction as map disjunction. Two maps are defined as being disjoint when their domains are disjoint.

Definition 6.7 ► *Map disjunction*

$$h_0 \perp h_1 \equiv \text{dom}\ h_0 \cap \text{dom}\ h_1 = \emptyset$$

Setting heap disjunction as map disjunction and the zero of our algebra as the empty map satisfies the first two axioms of our locale. Similarly, we define heap addition as map addition. Map addition combines two maps, and when they both maps have an entry for a particular value, the entry of the second map is used (commonly referred to as right-override).

Definition 6.8 ► *Map addition*

$$h_0 +\!+ h_1 = (\lambda x.\ \text{case}\ h_1\ x\ \text{of None} \Rightarrow h_0\ x \mid \text{Some}\ y \Rightarrow \text{Some}\ y)$$

We note that while the third of our axioms is clearly true, map addition is not always commutative. Thankfully, it is commutative when the maps are disjoint which satisfies our fourth axiom. Map addition is always associative, and so the fifth axiom is satisfied.

The sixth axiom is the only potentially unexpected axiom, and is in fact a stronger version of the corresponding axiom in Calcagno et al. (2007). It specifies (assuming that h_0 and h_1 are disjoint, and can thus be added) that if h is disjoint to $h_0 + h_1$, then h is disjoint to both h_0 and h_1 (and vice versa). This axiom is used to show that separation conjunction is associative, amongst other things.

Of course, these axioms allow more exotic heap structures to be shown to be separation algebras. The use of map addition and map disjunction in this section is meant only as an aid to intuition.

If we provide our own definitions for heap disjunction, heap addition, and the zero of the algebra, and show that they satisfy these 6 axioms, then we get a separation logic that comes with a number of things for free: definitions, such as separation conjunction; properties, such as the fact that separation conjunction is associative and commutative; and tactics, such as those that do frame calculation automatically. Some of these are part of the scope of this thesis and others are

done in collaboration with others. One of the things that we add as part of this thesis is support for dealing with lists and sets of separation conjunctions, and using these to reason about mapping a monadic operation over a list, as explained in Section 6.4. A number of further properties of these structures, developed for this thesis, which are true for any separation logic, are explored in Section 7.6 where they are used.

6.4 The properties of a separation logic

Separation logics have a number of properties common to them. By proving such properties about an abstract separation logic *once*, then we can have all of these properties "for free" for a specific separation logic if we show that our structure forms a separation algebra. The following properties were proven as part of this thesis, but are true for any instantiation of the abstract separation logic.

When defining propositions using separation logics, we often want to specify a property is true for a number of objects. For example, consider that we wish to specify that a range of addresses contain Untyped objects. We can express this by mapping the predicate $obj\_id \mapsto_{\mathsf{obj}} \mathsf{Untyped}$ over a list of object identifiers (*obj_ids*), giving us a list of separation predicates. We then fold this list using separation conjunction onto the empty predicate $\square$ (where $\square \equiv \lambda h.\ h = 0$, that is, the predicate stating that the heap is empty).

$$\mathsf{foldl}\ (\lambda P\ Q.\ P \wedge^* Q)\ \square\ (\mathsf{map}\ (\lambda obj\_id.\ obj\_id \mapsto_{\mathsf{obj}} \mathsf{Untyped})\ obj\_ids)$$

As such a style of predicate is common, we define a lifted separation conjunction that does this fold for us.

$$\bigwedge^* Ps \equiv \mathsf{foldl}\ (\lambda P\ Q.\ P \wedge^* Q)\ \square\ Ps$$

◀ **Definition 6.9**
Lifted separation conjunction for lists

This lets us express our above predicate about untyped objects as follows:

$$\bigwedge^* \mathsf{map}\ (\lambda obj\_id.\ obj\_id \mapsto_{\mathsf{obj}} \mathsf{Untyped})\ obj\_ids$$

Sometimes it is more convenient to map a separation predicate over a set rather than a list. To do so, we define the following predicate, which uses Isabelle/HOL's inbuilt mechanisms to fold a function over a set, which does precisely what we require.

Definition 6.10 ▶ *Lifted separation conjunction for sets*

$$\bigwedge^*_{x \in A} P\, x \equiv \mathsf{fold}\ (\lambda x\ Q.\ P\, x \wedge^* Q)\ \square\ A$$

While the above definition looks a little terse, it makes more sense when we realise that we can rewrite our lifted separation conjunction on the list map P xs as shown in Lemma 6.7.

Lemma 6.7 ▶ *Lifted separation conjunction on mapped lists*

$$\bigwedge^* \mathsf{map}\ P\ xs = \mathsf{foldl}\ (\lambda Q\ x.\ P\, x \wedge^* Q)\ \square\ xs$$

Assuming that our list is distinct (which they generally are for separation predicates), we can convert between the list and set forms of lifted separation conjunction as shown in Lemma 6.8. This allows us to use whichever form is more convenient. Recall that in this thesis we use the notation $x \in xs$ for both list and set membership.

Lemma 6.8 ▶ *Converting lifted separation conjunction on lists and sets*

$$\frac{\mathsf{distinct}\ xs}{\bigwedge^* \mathsf{map}\ P\ xs = (\bigwedge^*_{x \in xs} P\, x)}$$

To simplify some of our rules, we introduce the following shorthand when our set is defined using a predicate.

Definition 6.10 ▶ *Lifted separation conjunction for sets defined by predicates*

$$\bigwedge^*_{x\,|\,f\,x} P\, x \equiv \bigwedge^*_{x \in \{x\,|\,f\,x\}} P\, x$$

These definitions give us a convenient way of expressing the predicates used by the proof of the system initialiser, such as those which state that each object in a specification has been correctly created or is correctly initialised.

Reasoning about monadic maps

The great power of using a separation logic is that it allows us to easily say that an operation is local, and so we are able to chain together operations that modify distinct objects. This allows us to reason conveniently about monadically mapping an operation over a list of objects, as expressed in the following theorem.

Theorem 6.1 ▶ *Mapping monadic operations over lists*

$$\frac{\forall R'\ x.\ x \in xs \longrightarrow \{P\, x \wedge^* I \wedge^* R'\}\ f\ x\ \{Q\, x \wedge^* I \wedge^* R'\}}{\{\bigwedge^* \mathsf{map}\ P\ xs \wedge^* I \wedge^* R\}\ \mathsf{mapM}\ f\ xs\ \{\bigwedge^* \mathsf{map}\ Q\ xs \wedge^* I \wedge^* R\}}$$

Proof: This rule is proven by induction on xs. The base case where xs is the empty list is trivially true.

In the inductive case, we set xs to be $x' \cdot xs'$. Since f is local, we can know that each execution of f will affect only part of the state described by $P\ x'$, and so the rest of the state, that is $\bigwedge^*$ map $P\ xs' \wedge^* R$, can be covered by the arbitrary predicate R' in the assumptions, and is thus unchanged. ■

We can also express this theorem in the set form of lifted separation conjunction that we described above, assuming the list xs is distinct. This is the form of the rule we use most commonly in the proofs.

$$\frac{\mathsf{distinct}\ xs \qquad \forall R\ x.\ x \in xs \longrightarrow \{P\ x \wedge^* I \wedge^* R\}\ f\ x\ \{Q\ x \wedge^* I \wedge^* R\}}{\{\bigwedge^*{}_{x \in xs}\ P\ x \wedge^* I \wedge^* R\}\ \mathsf{mapM}\ f\ xs\ \{\bigwedge^*{}_{x \in xs}\ Q\ x \wedge^* I \wedge^* R\}}$$

◀ **Corollary 6.1** *Mapping monadic operations over lists*

The above theorem is one of the important theorems in this thesis, and one that makes this thesis possible. It allows us to reason about the many loops in the system initialisation specification in a simple and modular way. It is the reason that we use separation logic in this thesis.

6.5 Defining a fine-grained separation logic for capDL

When constructing a separation logic, one design decision that needs to be made is one of granularity. The separation logic that we defined in Section 6.2 has an object-level granularity—it allows us to reason that the setting of an object is a local operation, but it does not allow us to extend this reasoning to the individual capability level.

Because our system initialiser does not set objects to their initialised state atomically, but generally set objects capability by capability, we define a separation logic for this thesis at a finer level of granularity. To do so, we break the capDL objects up into various *components*, and store these components in a component heap, which is a mapping from *cdl_object_id* × *cdl_component_id* to *cdl_component*, as illustrated in Figure 6.1.

datatype *cdl_component_id* = Fields | Slot *nat*
datatype *cdl_component* = CDL_Object *cdl_object* | CDL_Cap (*cdl_cap option*)

These components of an object are either the *fields* of an object, such as the intent of a TCB or the size of a CNode, or the contents of an individual *capability slot* of an object (that is, a capability). Rather than creating a new type for the fields of an object, we store the fields of an object in a capDL object for which we simply ignore the capability slots. When storing a capability in a component, since

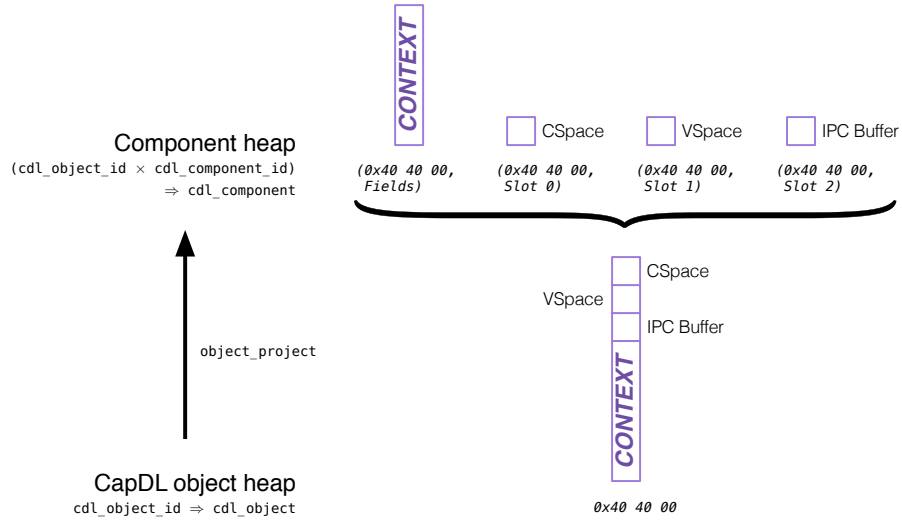


Figure 6.1: Lifting the capDL object heap to the component heap.

objects store capabilities in a partial mapping from capability slots to capabilities, it is easier to make a capability stored in a component optional.

Another alternative way to achieve a fine level of granularity would be to augment the capDL state with ownership information stating which parts of the objects in the heap are present, as we presented in an earlier version of this work (Klein et al., 2012). We found that such an approach gave us much more complex definitions of heap addition, which unnecessarily complicated proofs of the frame rules presented in Section 6.7, with no advantage of the expressiveness of our logic.

As well as lifting the object heap, we also lift the IRQ table (which translates IRQ numbers to the object identifier pointing to the corresponding IRQ Node) into our separation state. To do so, we transform the IRQ table from a total function, to a partial mapping.

Our lifted separation state contains these two heaps—the heap containing the object components, and the heap containing the IRQ table. The former is a partial function of type $cdl\_object\_id \times cdl\_component\_id \Rightarrow cdl\_component$, and the latter $cdl\_irq \Rightarrow cdl\_object\_id\ option$. The definition of heap disjunction and heap addition is the map disjunction and map addition of each of these heaps respectively.

Definition 6.11 ► sep_state_add

$$\text{sep\_state\_add}\ (\text{SepState}\ comp\_heap_L\ irq\_heap_L)\ (\text{SepState}\ comp\_heap_R\ irq\_heap_R) \equiv \text{SepState}\ (comp\_heap_L +\!\!+ comp\_heap_R)\ (irq\_heap_L +\!\!+ irq\_heap_R)$$

Definition 6.12 ► sep_state_disj

$$\text{sep\_state\_disj}\ (\text{SepState}\ comp\_heap_L\ irq\_heap_L)\ (\text{SepState}\ comp\_heap_R\ irq\_heap_R) \equiv comp\_heap_L \perp comp\_heap_R \wedge irq\_heap_L \perp irq\_heap_R$$

Because our component heap and our IRQ heap are standard heaps, we can define our separation state addition and disjunction as the addition and disjunction of both of the heaps, using standard map addition and disjunction, just like in Section 6.3, thus satisfying the six axioms for the same reasons.[7]

This state lifting affords other advantages, such as allowing us to easily lift only the parts of the capDL state that we wish to reason about, such as the object heap and IRQ table, but not the capability derivation tree nor the ASID numbers of objects. To lift the object heap out of the capDL state, before we break each object into its component pieces, we "clean" out two parts of information in them, using object_clean. The two parts of the object state that we remove when cleaning are the two parts of the state that we do not reason about in this proof, namely the ASIDs for memory capabilities (page directories, page tables, and frames), and the intent of a TCB. These are removed because of technical limitations—the former as our proof does not guarantee specific ASID numbers for memory objects[8], and the latter because the setting of the intent of an object is presently a non-local operation as the intents of different TCBs may overlap. As the intent of an object encodes the system call to be performed by an object, not guaranteeing the contents of a TCB's intent is not a practical limitation as a thread sets its intent before a system call.

◀ **Definition 6.13** object_clean

$$\mathsf{object\_clean} \equiv \mathsf{intent\_reset} \circ \mathsf{asid\_reset}$$

We use object_project to project an object from the capDL object heap into a component for the component heap. For capabilities, the component stores the relevant capability, while, for the fields of an object, we store an object with its capability slots wiped (as we wish to store only the fields of an object).

◀ **Definition 6.14** object_project

$$\begin{aligned}&\mathsf{object\_project}\ \mathit{comp\_id}\ \mathit{object} \equiv \mathsf{case}\ \mathit{comp\_id}\ \mathsf{of}\\ &\mathsf{Fields} \Rightarrow \mathsf{CDL\_Object}\ (\mathsf{object\_wipe\_slots}\ (\mathsf{object\_clean}\ \mathit{object}))\\ &\mid \mathsf{Slot}\ \mathit{slot} \Rightarrow \mathsf{CDL\_Cap}\ (\mathsf{object\_slots}\ (\mathsf{object\_clean}\ \mathit{object})\ \mathit{slot})\end{aligned}$$

[7] The algebraic structure of our separation logic is, in fact, a member of a stronger separation algebra, namely a cancellative separation algebra, meaning that it satisfies an additional axiom to the six already mentioned, namely $[\![h_o + h = h_1 + h; h_o \#\# h; h_1 \#\# h]\!] \Longrightarrow h_o = h_1$.

[8] Our system initialisation algorithm assigns ASIDs to virtual address spaces from its ASID Pool.

To project a single capability, we can use cap_project.

Definition 6.15 ▶ cap_project

$$\text{cap\_project } \mathit{cap} \equiv \text{CDL\_Cap } (\text{Some } (\text{reset\_cap\_asid } \mathit{cap}))$$

As expected, projecting a capability of an object is equivalent to projecting the capability directly.

Lemma 6.9 ▶ *Object project single capability*

$$\frac{\text{object\_slots } \mathit{object}\ \mathit{slot} = \text{Some } \mathit{cap}}{\text{object\_project } (\text{Slot } \mathit{slot})\ \mathit{object} = \text{cap\_project } \mathit{cap}}$$

To lift the object component heap, we lift each object individually and to lift the IRQ heap, we turn the IRQ table from total function to a partial function, as shown below.

Definition 6.16 ▶ sep_state_projection

$$\begin{aligned}&\text{sep\_state\_projection } s \equiv \\ &\text{SepState } (\lambda(\mathit{obj\_id}, \mathit{comp\_id}).\ \text{map } (\text{object\_project } \mathit{comp\_id})\ (\text{cdl\_objects } s\ \mathit{obj\_id})) \\ &\qquad (\lambda \mathit{irq}.\ \text{Some } (\text{cdl\_irq\_node } s\ \mathit{irq}))\end{aligned}$$

To lift a separation predicate P to operate on the capDL state, we use the notation $<P>\ s \equiv P\ (\text{sep\_state\_projection } s)$. To lift P to work on the system initialiser state (which embeds the capDL kernel state inside, as we explain in Section 7.4) we use the notation $\ll P \gg\ s \equiv P\ ((\text{sep\_state\_projection} \circ \text{kernel\_state})\ s)$. These predicates simply lift the capDL kernel state (or the system initialiser state) to a separation state and pass the separation state to the separation predicate P.

6.6 The arrows of our separation logic

With our separation logic defined, we then extend it by defining separation predicates to describe the existence of both whole objects, as well as the existence of individual components of an object (such as a single capability slot, or the fields of an object[9]). To do so, we define a number of *arrows* which state that part of an object is located at a particular location. Each of these arrows state that the heap contains *only* the specified object, component, or IRQ table entry. The first arrow that we introduce, $\mathit{obj\_id} \mapsto_{\text{obj}} \mathit{object}$, states that the lifted state contains only the object *object* at location *obj_id* in the object heap. This defines the same concept as the corresponding arrow we defined for our simple separation logic in Definition 6.3, but here we define our arrow on our lifted state.

[9] Our granularity goes down to the level of all of the fields or none of the fields. Given that these fields are generally set atomically, this level of granularity works well for our proofs.

◀ **Definition 6.17**
sep_map_obj

$$
\begin{aligned}
&\mathit{obj\_id} \mapsto_{\mathsf{obj}} \mathit{object} \equiv \\
&\lambda s.\ s = \mathsf{SepState} \\
&\qquad (\lambda(\mathit{obj\_id}', \mathit{comp\_id}). \\
&\qquad\quad \mathsf{if}\ \mathit{obj\_id}' = \mathit{obj\_id} \\
&\qquad\quad\ \mathsf{then\ Some}\ (\mathsf{object\_project}\ \mathit{comp\_id}\ \mathit{object})\ \mathsf{else\ None}) \\
&\qquad \mathsf{map\_empty}
\end{aligned}
$$

This definition states that the component heap consists only of the object *object*, as lifted from the capDL types to component heap, and that the IRQ table heap is empty. This definition makes the same constraints on the state that the corresponding arrow back in Section 6.2 does, but does so on the lifted separation state, rather than directly on the capDL object heap.

We introduce the predicate $\mathit{obj\_id} \mapsto_{\mathsf{fields}} \mathit{object}$ which states that the component heap contains only the component containing the object fields of *object*, at location $(\mathit{obj\_id}, \mathsf{Fields})$. This predicate dictates the type of the object, but states nothing about the capabilities.

◀ **Definition 6.18**
sep_map_fields

$$
\begin{aligned}
&\mathit{obj\_id} \mapsto_{\mathsf{fields}} \mathit{object} \equiv \\
&\lambda s.\ s = \mathsf{SepState} \\
&\qquad (\lambda(\mathit{obj\_id}', \mathit{comp\_id}). \\
&\qquad\quad \mathsf{if}\ \mathit{obj\_id}' = \mathit{obj\_id} \wedge \mathit{comp\_id} = \mathsf{Fields} \\
&\qquad\quad \mathsf{then\ Some}\ (\mathsf{object\_project}\ \mathit{comp\_id}\ \mathit{object})\ \mathsf{else\ None}) \\
&\qquad \mathsf{map\_empty}
\end{aligned}
$$

We introduce the predicate $(\mathit{obj\_id}, \mathit{slot}) \mapsto_{\mathsf{cap}} \mathit{cap}$ to specify that the component heap contains only the component with the capability *cap* stored at location $(\mathit{obj\_id}, \mathsf{Slot}\ \mathit{slot})$.

◀ **Definition 6.19**
sep_map_cap

$$
\begin{aligned}
&(\mathit{obj\_id}, \mathit{slot}) \mapsto_{\mathsf{cap}} \mathit{cap} \equiv \\
&\lambda s.\ s = \mathsf{SepState} \\
&\qquad (\lambda(\mathit{obj\_id}', \mathit{comp\_id}). \\
&\qquad\quad \mathsf{if}\ \mathit{obj\_id}' = \mathit{obj\_id} \wedge \mathit{comp\_id} = \mathsf{Slot}\ \mathit{slot} \\
&\qquad\quad\ \mathsf{then\ Some}\ (\mathsf{cap\_project}\ \mathit{cap})\ \mathsf{else\ None}) \\
&\qquad \mathsf{map\_empty}
\end{aligned}
$$

While the above predicate is useful for specifying the behaviour of an operation such as set_cap or a seL4 kernel API, it is often more convenient when describing the initialisation of an object to specify that a particular capability inside an object is present in the component heap, for which we provide the predicate

$(obj\_id, slot) \mapsto_{\mathsf{slot}} object$. Because this predicate constrains only the capabilities of an object, it constrains nothing about the type of the object.

Definition 6.20 ▶ sep_map_slot

$$
\begin{array}{l}
(obj\_id, slot) \mapsto_{\mathsf{slot}} object \equiv \\
\lambda s.\ s = \mathsf{SepState} \\
\quad (\lambda(obj\_id', comp\_id). \\
\quad\quad \mathsf{if}\ obj\_id' = obj\_id \wedge comp\_id = \mathsf{Slot}\ slot \\
\quad\quad \mathsf{then\ Some}\ (\mathsf{object\_project}\ comp\_id\ object)\ \mathsf{else\ None}) \\
\quad \mathsf{map\_empty}
\end{array}
$$

Of course, if the object located at address *obj_id* has a capability *cap* in the capability slot *slot*, then the predicate $(obj\_id, slot) \mapsto_{\mathsf{cap}} cap$ should be equivalent to the predicate $(obj\_id, slot) \mapsto_{\mathsf{slot}} object$. We prove that this is indeed the case in the following lemma.

Lemma 6.10 ▶ *Cap predicate equivalence*

$$\frac{\mathsf{object\_slots}\ object\ slot = \mathsf{Some}\ cap}{(obj\_id, slot) \mapsto_{\mathsf{slot}} object = (obj\_id, slot) \mapsto_{\mathsf{cap}} cap}$$

Proof: This is proven by expanding the definitions of the two predicates and using Lemma 6.9. ■

The predicate $obj\_id \mapsto_{\mathsf{slots}} object$ states that *all* of the capabilities of an object are present in the component heap.

Definition 6.21 ▶ sep_map_slots

$$
\begin{array}{l}
obj\_id \mapsto_{\mathsf{slots}} object \equiv \\
\lambda s.\ s = \mathsf{SepState} \\
\quad (\lambda(obj\_id', comp\_id). \\
\quad\quad \mathsf{if}\ obj\_id' = obj\_id \wedge comp\_id \neq \mathsf{Fields} \\
\quad\quad \mathsf{then\ Some}\ (\mathsf{object\_project}\ comp\_id\ object)\ \mathsf{else\ None}) \\
\quad \mathsf{map\_empty}
\end{array}
$$

To specify that all of the empty capability slots of an object are present in the component heap, we use the predicate $obj\_id \mapsto_{\mathsf{empty}} object$.

Definition 6.22 ▶ sep_map_empty

$$
\begin{array}{l}
obj\_id \mapsto_{\mathsf{empty}} object \equiv \\
\lambda s.\ s = \mathsf{SepState} \\
\quad (\lambda(obj\_id', comp\_id). \\
\quad\quad \mathsf{if}\ obj\_id' = obj\_id \wedge comp\_id \in \mathsf{Slot} \setminus (\mathsf{UNIV} - \mathsf{dom}\ (\mathsf{object\_slots}\ object)) \\
\quad\quad \mathsf{then\ Some}\ (\mathsf{object\_project}\ comp\_id\ object)\ \mathsf{else\ None}) \\
\quad \mathsf{map\_empty}
\end{array}
$$

Finally, we provide the predicate $irq \mapsto_{\mathsf{IRQ}} obj\_id$ which states that the IRQ table consists of the entry where the IRQ number *irq* is mapped to the object identifier *obj_id*.

$$irq \mapsto_{\mathsf{IRQ}} obj\_id \equiv \lambda s.\ s = \mathsf{SepState\ map\_empty}\ [irq \mapsto obj\_id]$$

◀ **Definition 6.23**
sep_map_irq

Arrow decomposition

When designing a separation logic, the question of granularity is an important one. Sometimes we wish to use an object-level granularity, and other times we wish to specify the contents of a specific capability slot. One of the strengths of our separation logic is that we can do both of these. We decompose a larger predicate, such as $obj\_id \mapsto_{\mathsf{obj}} object$ into smaller predicates, such as $obj\_id \mapsto_{\mathsf{fields}} object$ and $obj\_id \mapsto_{\mathsf{slots}} object$. To do so, we first show that we can decompose *any* arrow that relates to a set of components (such as the fields or the capability slots of an object) into two arrows talking about part of these components.

$$\frac{comp\_ids_L \cap comp\_ids_R = \emptyset}{\begin{array}{l}(\lambda s.\ s = \mathsf{SepState} \\ \quad (\lambda(obj\_id', comp\_id). \\ \quad\quad \mathsf{if}\ obj\_id' = obj\_id \wedge comp\_id \in comp\_ids_L \cup comp\_ids_R \\ \quad\quad \mathsf{then\ Some}\ (\mathsf{object\_project}\ comp\_id\ object)\ \mathsf{else\ None}) \\ \quad \mathsf{map\_empty}) = \\ ((\lambda s.\ s = \mathsf{SepState} \\ \quad (\lambda(obj\_id', comp\_id). \\ \quad\quad \mathsf{if}\ obj\_id' = obj\_id \wedge comp\_id \in comp\_ids_L \\ \quad\quad \mathsf{then\ Some}\ (\mathsf{object\_project}\ comp\_id\ object)\ \mathsf{else\ None}) \\ \quad \mathsf{map\_empty})\ \wedge^* \\ (\lambda s.\ s = \mathsf{SepState} \\ \quad (\lambda(obj\_id', comp\_id). \\ \quad\quad \mathsf{if}\ obj\_id' = obj\_id \wedge comp\_id \in comp\_ids_R \\ \quad\quad \mathsf{then\ Some}\ (\mathsf{object\_project}\ comp\_id\ object)\ \mathsf{else\ None}) \\ \quad \mathsf{map\_empty}))\end{array}}$$

◀ **Lemma 6.11**
Arrow decomposition

Proof: To prove this, we expand the definition of separation conjunction and divide our state into the parts relating to each of the components. ■

Using Lemma 6.11, we can then show that we can decompose any of our arrows into smaller arrows, for example, we decompose the predicate that states that a whole object is present into a predicate that states that each of the various parts of this object are present.

Theorem 6.2 ► *Object decomposition*

$$\frac{\mathsf{dom}\,(\mathsf{object\_slots}\ object) = slots \qquad \mathsf{finite}\ slots}{\begin{array}{l} obj\_id \mapsto_{\mathsf{obj}} object = \\ (obj\_id \mapsto_{\mathsf{fields}} object \wedge^* \bigwedge^*_{slot \in slots} (obj\_id, slot) \mapsto_{\mathsf{slot}} object \wedge^* obj\_id \mapsto_{\mathsf{empty}} object) \end{array}}$$

Proof: To prove this rule, we induct over the slots and then use Lemma 6.11 for each of our decompositions. ■

These arrows and their decomposition gives us a framework for being able to describe the state of the capDL kernel that can be used for reasoning about a low-level kernel API, which is concerned with moving particular capabilities between particular capability slots, right up to being able to say that a set of objects with particular properties are present in the state.

This reasoning framework is highly flexible as we will see in Chapter 7, and the decomposability of our predicates allows us to decompose our proofs in the obvious manner.

6.7 The frame rule

With both our separation logic and the arrows required to specify the state defined, we can now show that our operations such as set_object or set_cap are local.

The following lemma shows that set_object is local. It is equivalent to the version we showed in our simple separation logic in Section 6.2, namely Lemma 6.5, except that it is defined on the more complex separation logic that we use for this thesis (which is why we lift the predicates to work on the separation state using <P>).

Lemma 6.12 ► *Set object*

$$\begin{array}{l} \{\lambda s.\ <obj\_id \mapsto_{\mathsf{obj}} - \wedge^* R> s\} \\ \ \mathsf{set\_object}\ obj\_id\ object \\ \{\lambda s.\ <obj\_id \mapsto_{\mathsf{obj}} object \wedge^* R> s\} \end{array}$$

The arbitrary R in this lemma is what we use to show that set_object only affects the object located at *obj_id*. Traditionally, to show a rule such as this, we use the *frame rule*, as illustrated in Theorem 6.3.

Theorem 6.3 ► *Frame rule*

$$\frac{\{P\}\ f\ \{Q\}}{\{P \wedge^* R\}\ f\ \{Q \wedge^* R\}}$$

where no variable occurring free in R is modified by f.

The frame rule states that if $\{P\}\ f\ \{Q\}$, then adding more state will not change the behaviour of f. This is true as long as there are no free variables in R that are modified by f.

To use the frame rule, we require a memory safe language. The formalised semantics of our language need to be such that:

1. adding more memory does not affect the behaviour of a program,
2. a program fails if it does not have the resources that it requires.

We formalise our language's semantics of a program f executing on a state s as $f\ s$, where our program either succeeds in a valid state $\mathsf{Valid}\ s$, or fails with a state *Fail*.

We formalise the first property of adding more memory by requiring that if a program succeeds on a heap, then it will succeed on a larger heap (*safety monotonicity*).

$$\frac{f\ s_0 \neq \mathsf{Abort} \qquad s_0 \perp s_1}{f\ (s_0 +\!\!+ s_1) \neq \mathsf{Abort}}$$

◀ **Theorem 6.4** *Safety monotonicity*

We formalise the second property by saying that if a program executes on a smaller and a larger state, then execution on the larger state can be tracked back to the smaller state (*frame property*).

$$\frac{f\ s_0 \neq \mathsf{Abort} \qquad f\ (s_0 +\!\!+ s_1) = \mathsf{Valid}\ s' \qquad s_0 \perp s_1}{\exists {s_0}'.\ s' = {s_0}' +\!\!+ s_1 \wedge f\ s_0 = {s_0}'}$$

◀ **Theorem 6.5** *Frame property*

Yang and O'Hearn (2002) showed that the frame rule is equivalent to showing that our language possesses these two properties. Sadly, for our shallow embedding of a language, such as the formalisation of our system initialiser, we do not have these properties (Klein et al., 2012). Thankfully, proving the conclusion of the frame rule directly for our leaf functions is not onerous, and once we have these proven for the leaf operations, such as set_cap or set_object, then we can have them for free for all other operations that are built on top of these leaf functions. This approach is not new and is used on shallow embeddings of languages by Kolanski (2011).

For this thesis, we proved the following frame rules shown in Figure 6.2, which were then used by others to develop a formalised API for capDL, as explained in Section 7.4.

These rules are proven by expanding the definitions of the respective operations, as well as our separation logic definitions. The definition of separation

$\{\lambda s.\ <obj\_id \mapsto_{\mathsf{obj}} - \wedge^* R> s\}$
set_object *obj_id object*
$\{\lambda s.\ <obj\_id \mapsto_{\mathsf{obj}} object \wedge^* R> s\}$

$\{\lambda s.\ <ptr \mapsto_{\mathsf{cap}} - \wedge^* R> s\}$
set_cap *ptr cap*
$\{\lambda s.\ <ptr \mapsto_{\mathsf{cap}} cap \wedge^* R> s\}$

$\{\lambda s.\ <obj\_id \mapsto_{\mathsf{fields}} \mathsf{Tcb}\ tcb \wedge^* R> s\}$
update_thread_fault_endpoint *obj_id fault_ep*
$\{\lambda s.\ <obj\_id \mapsto_{\mathsf{fields}} \mathsf{Tcb}\ (tcb(\!|\ \mathrm{cdl\_tcb\_fault\_endpoint} := fault\_ep\ |\!)) \wedge^* R> s\}$

$\{\lambda s.\ <obj\_id \mapsto_{\mathsf{obj}} - \wedge^* R> s\}$
create_object *obj_id obj*
$\{\lambda s.\ <obj\_id \mapsto_{\mathsf{obj}} obj \wedge^* R> s\}$

$\{\lambda s.\ <obj\_id \mapsto_{\mathsf{obj}} obj \wedge^* R> s\}$
detype_object *obj_id*
$\{\lambda s.\ <obj\_id \mapsto_{\mathsf{obj}} \mathsf{Untyped} \wedge^* R> s\}$

Figure 6.2: Frame rules for the leaf functions of capDL.

conjunction states that we need to break our state up into two pieces, the first containing the part of the state we are modifying (such as the component with the capability that set_cap modifies) and the second piece containing the rest of the state.

We can then use these "frame rules" we have developed for our leaf functions to prove the "frame rule" for other operations that build on top of these leaf functions. For example, to reason about swap_cap we use our rule for set_cap. We continue this process to slowly build up separation logic rules for the seL4 API, that will be shown in Section 7.4.

Since the system initialiser does not *detype* objects, we do not require the rule for the detyping of objects for this thesis, but it is included here to show the completeness of the separation logic. In capDL, objects are never really created or destroyed, but are instead retyped from *untyped* objects, and then can be detyped back into untyped objects, modelling the behaviour of the seL4 kernel which deals only with a finite memory. This model is similar to the destruction model we present in Chapter 4 (as should be expected). This model not only avoids the issues of identifier reuse described in Section 4.2, but the lack of object destruction avoids any issues in the separation logic that come from owning empty parts of the state, thus simplifying the logic.

6.8 Conclusion

The separation logic we have presented in this chapter is designed to allow us to prove that each step of our system initialiser algorithm brings us one step closer to having an initialised system, whilst also proving that each step does not undo any of the work done by previous steps.

By allowing reasoning about parts of the heap, separation logic allows us to easily specify exactly which objects are modified by the system initialiser and which objects remain unchanged. By designing a separation algebra with the right level of granularity we can specify exactly how individual components of an object are transformed.

In the next chapter we use this separation logic to both specify what it means for a system to be correctly initialised and to prove that our system initialiser algorithm will correctly initialise a system. The style of reasoning afforded by a separation logic fits well with the highly loop driven model of the system initialiser and is key to its proof.

Chapter Summary

- Separation logics allow *local reasoning* to take place on *separate parts* of an object heap.
- This local reasoning allows us to reason both about what an operation *does* and what it *does not* do.
- Rather than creating yet another separation logic from scratch, we reuse an abstract separation logic developed by Klein et al. (2012).
- Further, we extended the abstract separation logic of Klein et al. (2012) to allow us to fold separation logic predicates over list and sets, and to be able to reason about monadic loops (mapM).
- We have defined a fine-grained separation logic, designed to allow us to reason about the individual steps of system initialisation, such as setting each capability slot individually. This separation logic is defined on a *separation logic heap* consisting of *object components* and *IRQ table entries* that we lift out of the capDL state.
- We have defined a number of predicates for specifying the presence of objects, capabilities, object fields, and IRQ table entries in the separation logic heap, and have shown how we can *decompose* larger predicates about whole objects into predicates about the individual components of an object. This decomposability of our separation predicates allows us to decompose our proofs.

Chapter 7

Correctness of system initialisation

The aim of this thesis is to develop a framework for the correct initialisation of capability-based computer systems. To do this, we have developed a formal algorithm for system initialisation, explained in Chapter 5, and a proof framework for reasoning about such systems using separation logic, as introduced in Chapter 6. We have found separation logic convenient for both expressing what it means for a system to be correctly initialised and proving that it is indeed correctly initialised.

The correctness statement for the system initialiser states that, if the boot information provided by the kernel is correct, then at the end of the initialisation, all objects and all hardware interrupts in the system either belong to the initialiser itself (and are inactive), or are initialised in conformance with the given capDL specification, as illustrated in Figure 7.1. We give the formal definition of this fact in Section 7.3. The system initialiser, the proof, and the given specification are all defined using the capDL kernel model which simplifies reasoning.

In this chapter we dive into the proof of correctness for the formal algorithm of our system initialiser, and unpack what the above statement of correct initialisation means.

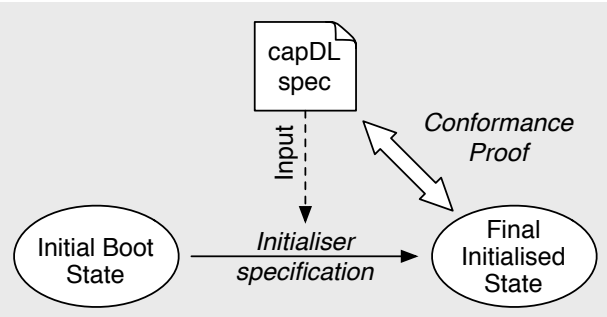


Figure 7.1: Overview of initialiser correctness proof.

- We define what it means for the individual objects in a system to be correctly initialised in Section 7.1.
- We explain in Section 7.2 how we specify the data structures of the initialiser itself—its capability space, the capabilities it requires, and the capabilities it creates.
- We provide the formal definition of what it means for a system to be correctly initialised in Section 7.3.
- We explain in Section 7.4 how we build this proof on a formalised API for seL4 developed by Boyton et al. (2013). This formalised API is built on the separation logic we constructed in Chapter 6.
- We explain what it means for a capDL specification to be well-formed, and list the assumptions of the proof in Section 7.5.
- We explain how we decompose the proof in Section 7.6, drilling in detail into the creation of IRQ Handler capabilities and the initialisation of capability spaces as representative examples.
- Finally we conclude with our experience of using separation logic for these proofs in Section 7.7.

Parts of this chapter have previously appeared in Boyton et al. (2013).

7.1 Correct object initialisation

In this section, we define precisely what it means for an object, or a hardware interrupt, to be correctly initialised.

In capDL specifications, systems are described as a mapping from object identifiers to objects. These identifiers of objects in a capDL specification are treated by the system initialiser as simply names—in a textual description they are a string naming the object, whereas in the C implementation, they are ascending numbers starting at zero.

The initialiser creates each of the objects specified in a capDL specification at particular physical memory addresses. This produces a natural injection φ between the object identifiers in a capDL specification *spec* and the physical memory addresses of the initialised system. The injection φ captures the subtlety that the kernel decides physical memory addresses at runtime.

For a system to be correctly initialised, each one of the objects in the capDL specification should exist in the initialised system, at a physical memory address, with the capabilities of each object pointing to the correct objects (which will each

also be located at physical memory addresses). More formally, if we consider a capDL specification as a graph of objects pointing to other objects, then φ can be used to define an injective homomorphism, or *monomorphism*, between the given capDL specification *spec* and the kernel state after system initialisation. This monomorphism is simply the renaming of object identifiers in the specification with the corresponding physical memory addresses, as shown in Figure 7.2. The reason that this is a monomorphism and not an isomorphism (that is, a bijective homomorphism), is that the kernel state will include the objects that belong to the initialiser itself, which do not appear in the capDL specification and so it cannot be bijective.

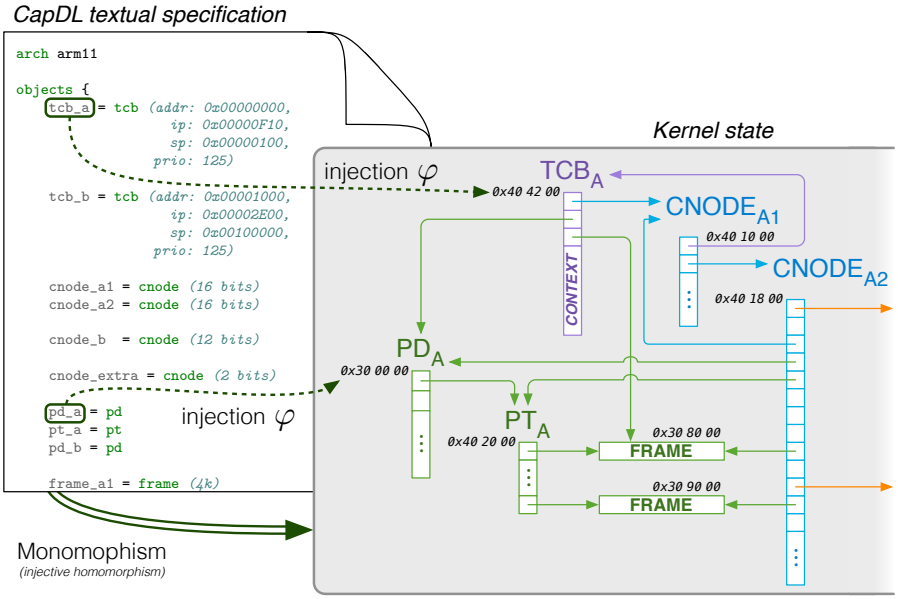


Figure 7.2: In a correctly initialised system, there is an injection φ between object identifiers in the given capDL specification and the physical memory addresses of the running kernel state. This injection φ defines a monomorphism between the two.

We define this monomorphism using two different predicates. Firstly, we define the predicate object_initialised that specifies that a "real" object is correctly initialised (along with the capabilities stored within these objects). Secondly, we define the predicate irqs_initialised that specifies that an IRQ Node is correctly initialised.[1]

The predicate object_initialised, defined in Definition 7.1, encodes the fact that, in a kernel state *s*, the object corresponding to *obj_id* in the capDL specification has been created as specified in the spec. More precisely, the predicate states that *obj_id* maps to a *kernel_obj_id* via the injection φ, and that, in the object heap

[1]We distinguished between "real" objects and IRQ Nodes in Section 5.3, and illustrated the types of objects supported by the system initialiser in Figure 5.6.

of s, *kernel_object_id* points to *spec_object* where all object identifiers have been renamed by φ—including those within the capabilities of each object, defined by spec2s. The function cdl_objects *spec* extracts the mapping from *obj_id* to object from *spec*.

Definition 7.1 ► object_initialised

$$\begin{aligned}&\text{object\_initialised } spec\ \varphi\ obj\_id \equiv\\ &\lambda s.\ \exists kernel\_obj\_id\ spec\_object.\ \varphi\ obj\_id = \text{Some } kernel\_obj\_id \wedge\\ &\quad (kernel\_obj\_id \mapsto_{\text{obj}} \text{spec2s } \varphi\ spec\_object)\ s \wedge\\ &\quad \text{cdl\_objects } spec\ obj\_id = \text{Some } spec\_object\end{aligned}$$

Similarly, the predicate irq_initialised, defined in Definition 7.2 specifies that the IRQ Node pointed to by an IRQ number is correctly initialised. Recall that IRQ numbers do not point to IRQ Nodes directly in capDL, but instead IRQ numbers are looked up in an IRQ table, which contains the object identifier of the corresponding IRQ Node.

We specify this correct initialisation of an IRQ number formally by stating that an IRQ Number *irq* maps to a *spec_irq_id* in the IRQ table of the capDL specification *spec*, which itself points to an IRQ Node *spec_irq_node*, and that *irq* maps to a *kernel_irq_id* in the IRQ table of the kernel state s, which itself points to an IRQ Node *spec_irq_node* where all object identifiers have been renamed by φ—including those within the capabilities of each object, defined by spec2s.

Definition 7.2 ► irq_initialised

$$\begin{aligned}&\text{irq\_initialised } spec\ \varphi\ irq\ s \equiv\\ &\exists kernel\_irq\_id\ spec\_irq\_node\ spec\_irq\_id.\ \varphi\ spec\_irq\_id = \text{Some } kernel\_irq\_id \wedge\\ &\quad (irq \mapsto_{\text{IRQ}} kernel\_irq\_id \wedge^{*} kernel\_irq\_id \mapsto_{\text{obj}} \text{spec2s } \varphi\ spec\_irq\_node)\ s \wedge\\ &\quad \text{cdl\_irq\_node } spec\ irq = spec\_irq\_id \wedge\\ &\quad \text{cdl\_objects } spec\ spec\_irq\_id = \text{Some } spec\_irq\_node\end{aligned}$$

As we recall from Section 5.3, our initialisation is a multistage process where all objects are first created and then initialised by their type. For this reason, we define the predicates object_empty and irq_empty to describe the state of these objects when they are first created.

The predicates object_empty and object_initialised are defined very similarly, the difference being that an empty object is not one with its capabilities renamed by spec2s, but one in a default state, as defined by the function object_default_state.

Definition 7.3 ► object_empty

$$\begin{aligned}&\text{object\_empty } spec\ \varphi\ obj\_id\ s \equiv\\ &\exists kernel\_obj\_id\ spec\_object.\ \varphi\ obj\_id = \text{Some } kernel\_obj\_id \wedge\\ &\quad (kernel\_obj\_id \mapsto_{\text{obj}} \text{object\_default\_state } spec\_object)\ s \wedge\\ &\quad \text{cdl\_objects } spec\ obj\_id = \text{Some } spec\_object\end{aligned}$$

Similarly, we define the predicate irq_empty, which defines the equivalent property for IRQ Nodes.

◀ **Definition 7.4**
irq_empty

$$\begin{aligned}&\mathsf{irq\_empty}\ spec\ \varphi\ irq\ s \equiv \\ &\exists kernel\_irq\_id\ spec\_irq\_node\ spec\_irq\_id.\ \varphi\ spec\_irq\_id = \mathsf{Some}\ kernel\_irq\_id \wedge \\ &\quad (irq \mapsto_{\mathsf{IRQ}} kernel\_irq\_id \wedge^* kernel\_irq\_id \mapsto_{\mathsf{obj}} \mathsf{object\_default\_state}\ spec\_irq\_node)\ s \wedge \\ &\quad \mathsf{cdl\_irq\_node}\ spec\ irq = spec\_irq\_id \wedge \\ &\quad \mathsf{cdl\_objects}\ spec\ spec\_irq\_id = \mathsf{Some}\ spec\_irq\_node\end{aligned}$$

7.2 State of the system initialiser

Recall that the system initialiser is the root task that runs first after the kernel has finished its own initialisation. The system initialiser has a number of objects (and components of these objects) that are unchanged throughout system initialisation, such as the fields of its TCB, or the IRQ Control capability the system initialiser uses to create IRQ Handler capabilities. We encode these using the predicate si_objects which we introduce below. These unchanged objects are used by the various rules that we define for each stage of system initialisation. These objects will all remain as part of the initialiser in the final initialised state.

The system initialiser also has a number of capabilities and objects that are changed throughout system initialisation. For example, the untyped capabilities store the free addresses they point to, which are consumed as objects are created. The system initialiser stores the original capabilities to each of these created objects, before moving them into the relevant capability spaces that it configures (as was illustrated in the figures back in Section 5.3).

In this section, we define predicates to describe these various states that the system initialiser is in throughout initialisation, concentrating particularly on the initial and final states of system initialisation.

Initial state of the system initialiser

The system initialiser has its own thread control block, capability space, virtual address space, and an empty ASID pool (for assigning ASIDs to page directories). We specify the layout of the parts of these objects that do not change using the predicate si_objects, shown in Definition 7.5. We specify, the capabilities and the fields of these objects, using the $obj\_id \mapsto_{\mathsf{fields}} object$ and $(obj\_id, slot) \mapsto_{\mathsf{cap}} cap$ notation from Section 6.6. The fields of the system initialiser's TCB, CNode, and ASID pool should be the default fields for objects of their size. The system initialiser's TCB should have a capability to its capability space and be set to running. The system initialiser's CNode should have a capability to itself, the IRQ Control

capability, and an ASID Pool. The system initialiser's ASID pool should have a number of entries in it, although their contents does not matter for these proofs and thus we leave it underspecified. These fields and capabilities specified by si_objects remain constant throughout system initialisation.

Definition 7.5 ► si_objects

$$
\begin{aligned}
&\mathsf{si\_objects} \equiv \mathsf{si\_tcb\_id} \mapsto_{\mathsf{fields}} \mathsf{Tcb\ default\_tcb} \wedge^* \\
&\mathsf{si\_cnode\_id} \mapsto_{\mathsf{fields}} \mathsf{CNode}\ (\mathsf{empty\_cnode\ si\_cnode\_size}) \wedge^* \\
&\mathsf{si\_ap\_id} \mapsto_{\mathsf{fields}} \mathsf{AsidPool\ empty\_asid} \wedge^* \\
&(\mathsf{si\_tcb\_id}, \mathsf{tcb\_cspace\_slot}) \mapsto_{\mathsf{cap}} \mathsf{si\_cspace\_cap} \wedge^* \\
&(\mathsf{si\_tcb\_id}, \mathsf{tcb\_pending\_op\_slot}) \mapsto_{\mathsf{cap}} \mathsf{RunningCap} \wedge^* \\
&(\mathsf{si\_cnode\_id}, \mathsf{seL4\_CapInitThreadCNode}) \mapsto_{\mathsf{cap}} \mathsf{si\_cnode\_cap} \wedge^* \\
&(\mathsf{si\_cnode\_id}, \mathsf{seL4\_CapIRQControl}) \mapsto_{\mathsf{cap}} \mathsf{IrqControlCap} \wedge^* \\
&(\mathsf{si\_cnode\_id}, \mathsf{seL4\_CapInitThreadASIDPool}) \mapsto_{\mathsf{cap}} \mathsf{AsidPoolCap\ si\_ap\_id\ si\_ap\_base} \wedge^* \\
&\bigwedge^*_{slot \mid slot < 2^{\mathsf{asid\_low\_bits}}} (\mathsf{si\_ap\_id}, slot) \mapsto_{\mathsf{cap}} -
\end{aligned}
$$

The system initialiser also has a number of other objects and capabilities given to it by seL4 after kernel initialisation which change during system initialisation and which are not specified by si_objects. These objects and capabilities that change, combined with the unchanged parts of objects described above, comprise the initial boot state of the system initialiser. We explain each of these objects and capabilities that change throughout system initialisation below.

The system initialiser has a number of untyped capabilities from which it creates the required objects, and it has a number of free capability slots to place the capabilities to each of the objects that it creates. The location of these untyped capabilities and free capability slots is encoded in the boot information passed to the kernel. We say that this boot information is valid if these capabilities at the locations specified by *bootinfo* and the free object identifiers for each of the untyped capabilities do indeed point to untyped objects. The locations of the untyped capabilities and free capability slots must fall within the size of the system initialiser's capability space. These untyped capabilities each contain a range of object identifiers that they cover (either those identifiers that are in use or those that could be used) and a range of object identifiers that are available for object creation. We require that the former is a superset of the latter. We also require that the free object identifiers pointed to by the untyped capabilities do not overlap[2]. These constraints are shown in Definition 7.7.

When seL4 initialises itself, it creates an IRQ Node for each IRQ number. For system initialisation we require only that the kernel has created IRQ Nodes for the

[2]This should be true for seL4 kernel initialisation, although proving that this is the case is outside the scope of this thesis.

IRQ numbers used in our given capDL specification *spec*.[3] We encode this using the separation logic predicate si_irq_nodes *spec*, defined in Definition 7.6. We do not need to know anything about the IRQ table itself, other than it maps from IRQ numbers to memory address containing empty IRQ Nodes.

Definition 7.6 si_irq_nodes

$$\begin{aligned}&\mathsf{si\_irq\_nodes}\ spec \equiv \lambda s.\ \exists k\_irq\_table.\\ &\qquad (\bigwedge^*_{irq \in \mathsf{used\_irqs}\ spec} (irq \mapsto_{\mathsf{IRQ}} k\_irq\_table\ irq \wedge^*\\ &\qquad\qquad k\_irq\_table\ irq \mapsto_{\mathsf{obj}} \mathsf{empty\_irq\_node}))\ s\end{aligned}$$

Finally, we also require that there are enough free capability slots for each of the objects (both for the original capabilities to the "real" objects that the system initialiser creates, and for the IRQ Handler capabilities pointing to the IRQ Nodes that the system initialiser initialises) and for the required duplicated capabilities (to TCBs and CNodes)[4]. Strictly speaking this is not part of having valid boot information, but considering the number of free capability slots can be set at compile time for seL4, we include it here in this definition. These requirements, namely the objects and capabilities required by the system initialiser, are formalised in the definition of valid_boot_info, shown in Definition 7.7.

Definition 7.7 valid_boot_info

$$\begin{aligned}&\mathsf{valid\_boot\_info}\ bootinfo\ spec \equiv\\ &\lambda s.\ \exists ut\_caps\ fstart\ fend\ ustart\ uend.\\ &\quad (\mathsf{si\_objects} \wedge^* \mathsf{si\_irq\_nodes}\ spec \wedge^*\\ &\quad \bigwedge^*_{(cptr,\ cap) \in \mathsf{zip}\ [ustart..uend-1]\ ut\_caps} (\mathsf{si\_cnode\_id}, cptr) \mapsto_{\mathsf{cap}} cap \wedge^*\\ &\quad \bigwedge^*_{cptr \in [fstart..fend-1]} (\mathsf{si\_cnode\_id}, cptr) \mapsto_{\mathsf{cap}} \mathsf{NullCap} \wedge^*\\ &\quad \bigwedge^*_{obj\_id \in (\bigcup_{cap \in ut\_caps} \mathsf{cap\_free\_ids}\ cap)} obj\_id \mapsto_{\mathsf{obj}} \mathsf{Untyped})\ s \wedge\\ &\quad |\mathsf{objects\_of}\ spec| + |\mathsf{real\_cnodes\_and\_tcbs\_of}\ spec| \leq fend - fstart \wedge\\ &\quad |ut\_caps| = uend - ustart \wedge \mathsf{distinct\_sets}\ (\mathsf{map}\ \mathsf{cap\_free\_ids}\ ut\_caps) \wedge\\ &\quad \bigwedge_{ut\_cap \in ut\_caps} \mathsf{is\_full\_untyped\_cap}\ ut\_cap \wedge\\ &\quad \bigwedge_{ut\_cap \in ut\_caps} \mathsf{well\_formed\_untyped\_cap}\ ut\_cap \wedge\\ &\quad \mathsf{bi\_untypes}\ bootinfo = (ustart, uend) \wedge \mathsf{bi\_free\_slots}\ bootinfo = (fstart, fend) \wedge\\ &\quad ustart < 2^{\mathsf{si\_cnode\_size}} \wedge uend \leq 2^{\mathsf{si\_cnode\_size}} \wedge uend \neq 0 \wedge\\ &\quad fstart < 2^{\mathsf{si\_cnode\_size}} \wedge fend \leq 2^{\mathsf{si\_cnode\_size}} \wedge fend \neq 0\end{aligned}$$

Intermediate states of the system initialiser

The system initialiser creates a number of capabilities during its operation—it creates a list of original capabilities to each new object it creates, it creates a list of duplicate capabilities to the TCBs and CNodes that it creates, and it also creates a

[3] Any other IRQ Nodes are unchanged by system initialisation.
[4] The duplication of these capabilities was described back in Section 5.3.

list of IRQ Handler capabilities. When the system initialiser wishes to access an object, or to hand out a capability to an object, it needs to know where the relevant capability is located. Because of this, it stores three data structures *orig_caps*, *dup_caps*, and *IRQ_caps*, as was explained in Section 5.3.[5]

The various rules in our proof need to know that these data structures hold the correct information. The data structure *orig_caps* maps object identifiers to the capability pointers (which in turn point to the corresponding kernel objects). We use the predicate si_cap_at φ *orig_caps spec obj_id*, defined in Definition 7.8, to specify that a single entry, namely *obj_id* in the mapping *orig_caps*, does indeed point to the right capability slot. (Note that the definition of si_cap_at can take in an arbitrary mapping from object identifiers to capability slots, not just *orig_caps*.)

We do this by specifying that in *orig_caps*, the object identifier *obj_id* points to a capability pointer *cptr*, and that *cptr* points to the default capability for the object, located at the physical memory address obtained by the injection φ. We also specify that this capability pointer *cptr* points to a capability slot within the system initialiser's CNode. Similarly, we use the same predicate si_cap_at to assert the correctness of the data structure *dup_caps* for a single object identifier *obj_id* by asserting si_cap_at φ *dup_caps spec obj_id*.

Definition 7.8 ▸ si_cap_at

si_cap_at φ *si_caps spec obj_id* $\equiv$
$\lambda s.\ \exists cptr\ object\ kernel\_object\_id.\ \varphi\ obj\_id = \mathsf{Some}\ kernel\_object\_id \wedge$
$((\mathsf{si\_cnode\_id}, cptr) \mapsto_{\mathsf{cap}} \mathsf{default\_cap\_to\_object}\ object\ kernel\_object\_id)\ s \wedge$
$\mathsf{cdl\_objects}\ spec\ obj\_id = \mathsf{Some}\ object \wedge$
$si\_caps\ obj\_id = \mathsf{Some}\ cptr \wedge cptr < 2^{\mathsf{si\_cnode\_size}}$

We define a similar separation logic predicate si_irq_cap_at to check the correctness of the data structure *irq_caps* for a single IRQ number, as defined in Definition 7.9. si_irq_cap_at *irq_caps spec irq* specifies that the data structure *irq_caps* points to a capability pointer *cptr* which points to an IRQ Handler capability (to the correct IRQ number) in the system initialiser's capability space.

Definition 7.9 ▸ si_irq_cap_at

si_irq_cap_at *si_irq_caps spec irq* $\equiv$
$\lambda s.\ \exists cptr.\ ((\mathsf{si\_cnode\_id}, cptr) \mapsto_{\mathsf{cap}} \mathsf{IrqHandlerCap}\ irq)\ s \wedge$
$si\_irq\_caps\ irq = \mathsf{Some}\ cptr \wedge cptr < 2^{\mathsf{si\_cnode\_size}}$

[5] In our C implementation, we declare these data structures as statically allocated global arrays, as shown in Section 5.4. This means that we do not need extra memory to store these data structures.

Final state of the system initialiser

When the system initialiser has finished creating all the required objects it will have handed out all of the original and IRQ Handler capabilities, but will still possess the duplicated capabilities that it created (as was illustrated in Figure 5.10). The objects and capabilities that the system initialiser possesses are specified by the predicate si_final_objects *spec* φ, expressed in Definition 7.10.

The objects of the system initialiser that remain after initialisation has completed are as follows. Firstly, the parts of the objects of the system initialiser that are unchanged, such as the root task's TCB and parts of the root CNode (all of which we encoded using si_objects), are all still present after initialisation. Secondly, the untyped capabilities will still exist (although with fewer available object identifiers) and the unused object identifiers of these untyped capabilities should point to untyped objects. Thirdly, the capability pointers pointing to the capability slots in the system initialiser's CNode which were used to hold the original capabilities and the IRQ Handler capabilities will now all be free (the first |objects_of *spec*| of the *free_cptrs*), as will the free slots that were not used (the last |objects_of *spec*| + |real_cnodes_and_tcbs_of *spec*| of the *free_cptrs*).[6] Finally, the duplicated capabilities the initialiser created to the CNodes and the TCBs will still be present.

◀ Definition 7.10 si_final_objects

$$
\begin{aligned}
&\mathsf{si\_final\_objects}\ spec\ \varphi \equiv \\
&\lambda s.\ \exists dup\_caps\ ut\_cptrs\ free\_cptrs\ ut\_caps\ all\_available\_ids. \\
&\quad (\mathsf{si\_objects} \wedge^* \\
&\quad \bigwedge^*_{(cptr,\ ut\_cap) \in \mathsf{zip}\ ut\_cptrs\ ut\_caps} (\mathsf{si\_cnode\_id}, cptr) \mapsto_{\mathsf{cap}} ut\_cap \wedge^* \\
&\quad \bigwedge^*_{obj\_id \in all\_available\_ids} obj\_id \mapsto_{\mathsf{obj}} \mathsf{Untyped} \wedge^* \\
&\quad \bigwedge^*_{cptr \in \mathsf{take}\ |\mathsf{objects\_of}\ spec|\ free\_cptrs} (\mathsf{si\_cnode\_id}, cptr) \mapsto_{\mathsf{cap}} \mathsf{NullCap} \wedge^* \\
&\quad \bigwedge^*_{cptr \in \mathsf{drop}\ (|\mathsf{objects\_of}\ spec| + |\mathsf{real\_cnodes\_and\_tcbs\_of}\ spec|)\ free\_cptrs} (\mathsf{si\_cnode\_id}, cptr) \mapsto_{\mathsf{cap}} \mathsf{NullCap} \wedge^* \\
&\quad \bigwedge^*_{obj\_id \in \mathsf{real\_cnodes\_and\_tcbs\_of}\ spec} \mathsf{si\_cap\_at}\ \varphi\ dup\_caps\ spec\ obj\_id)\ s
\end{aligned}
$$

[6] Recall from Section 5.3 that the capDL kernel model represents IRQ Nodes as CNodes of size zero, and so we use the term *real_cnodes spec* to describe the "real" CNodes (that is, not IRQ Nodes).

7.3 Top-level theorem

Using the predicates defined in Section 7.2, we can formally state the top-level theorem that we prove for system initialisation, as shown in Theorem 7.1.

Theorem 7.1 ► *Frame rule*

If well_formed *spec* and *obj_ids* = objects_of *spec* and distinct *obj_ids* then

$$\{\lambda s.\ \ll \text{valid\_boot\_info } \mathit{bootinfo}\ \mathit{spec} \wedge^* R \gg s\}$$
$$\text{init\_system } \mathit{spec}\ \mathit{bootinfo}\ \mathit{obj\_ids}$$
$$\{\lambda s.\ \exists \varphi.\ \ll \bigwedge^*_{\mathit{obj\_id} \in \text{real\_objects\_of } \mathit{spec}} \text{object\_initialised } \mathit{spec}\ \varphi\ \mathit{obj\_id} \wedge^*$$
$$\bigwedge^*_{\mathit{irq} \in \text{used\_irqs } \mathit{spec}} \text{irq\_initialised } \mathit{spec}\ \varphi\ \mathit{irq} \wedge^* \text{si\_final\_objects } \mathit{spec}\ \varphi \wedge^* R \gg s \wedge$$
$$\text{injective } \varphi \wedge \text{dom } \varphi = \mathit{obj\_ids}\}$$

This theorem states that, given a well_formed capDL specification (as explained in Section 7.5), the system initialiser, if it terminates, transforms an initial boot state described by *boot_info* into a state containing:

1. each "real" object in the specification correctly initialised,
2. each IRQ Node in the specification correctly initialised,
3. the data structures of the initialiser.[7]

Additionally, this theorem states that the mapping φ (from object identifiers in *spec* to the physical memory addresses where the objects are located) is injective and covers all specification objects (both "real" and IRQ Nodes). The "R" in this rule is the frame, as introduced in Section 6.7. It specifies that the rest of the kernel state, such as unused IRQ Nodes or the system initialiser's virtual address space, is unchanged during initialisation. In the remainder of this chapter we explain how we prove this result.

7.4 seL4 kernel semantics

Recall that our system initialiser specification, init_system, eventually calls functions in the seL4 API. For example, the operation init_cnode_slot, defined in Definition 5.7, invokes seL4 to move a capability using the seL4_CNode_Move system call. This means that to be able to prove the correctness of init_system we need formal specifications of the seL4 system calls that init_system uses.

One of the greatest risks when using formal specifications of software is that the specification does not meet requirements, is inconsistent, or does not match

[7] As explained in Section 7.2, the initialiser does not delete the capabilities that it duplicates, and so these also remain.

the code (Rushby, 2009). For example, when formal specifications for UNIX written in the Z specification language were later analysed, they were shown to contain numerous errors (Saaltink, 1997). In our work, we narrow the requirements gap by proving a high-level correctness statement. We additionally address the latter two hazards by not simply *assuming* a specification for the behaviour of seL4, but instead formally connect it to the capDL model of the seL4 kernel, which itself formally abstracts the seL4 binary as was illustrated in Figure 3.7. This capDL kernel model (Boyton et al., 2013) was developed in conjunction with this work and is a monadic specification of the behaviour of the kernel.

To join our proofs to this monadic specification of seL4, a formal specification of seL4's APIs were developed outside the scope of this work, which are formally connected to the seL4 kernel. Unlike most of the seL4 proofs, these API specifications describe the behaviour of seL4 from the perspective of a userspace program. These API specification proofs are of a different flavour to the other proofs about seL4, which were mostly concerned about global invariants and all possible, potentially malformed or malicious, inputs. Exercising the kernel API from a user-level proof, however, required a different perspective—given a specific "good" initial state for an API call, we need to know the effect of the API call on this state, and determine which other parts of the kernel state are and are not affected.

Separation logic proved a good match for this kind of specification, as the seL4 API calls tend to only modify small, localised parts of the system's state. These specifications were built using the separation logic implementation we developed for this thesis, as was the model of the seL4 system calls that we explain below.[8] The specifications themselves are outside the scope of this work, so we will not explain them in detail here.

The seL4 API specifications do not tell us anything about the kernel's internal data structures, but they give us the information we need for user-level proofs. The API specifications are typically large, around 30–50 lines each, because they capture the precise conditions needed for a specific kernel call to succeed. For example, the API specification of seL4_CNode_Move is shown in Figure 7.3, and is typical of many of the API specifications. By joining the proof of system initialisation to this verified abstraction of seL4, we are forced to deal with the intricacies of a low-level kernel API in our proofs.

The system model we developed for this work and used by these API specifications is somewhat simplistic—it assumes that only one thread in the system can

[8] In particular, the separation logic, the arrows, and the frame rule for the functions shown in Section 6.7 are all in scope of this thesis, as is the system model used by the seL4 kernel calls in the capDL kernel model. Using these to prove the API specifications such as Figure 7.3 is not part of the scope. The proof of the system initialiser, which uses these API specifications is in scope.

Theorem 7.2 ► *API specification of* seL4_CNode_Move

$$
\begin{aligned}
&\{\lambda s.\ \ll \mathsf{si\_tcb\_id} \mapsto_{\mathsf{fields}} tcb \wedge^* \\
&\quad (\mathsf{si\_tcb\_id}, \mathsf{tcb\_pending\_op\_slot}) \mapsto_{\mathsf{cap}} - \wedge^* \\
&\quad cnode\_id \mapsto_{\mathsf{fields}} \mathsf{CNode}\ (\mathsf{empty\_cnode}\ root\_size) \wedge^* \\
&\quad dest\_id \mapsto_{\mathsf{fields}} \mathsf{CNode}\ (\mathsf{empty\_cnode}\ dest\_size) \wedge^* \\
&\quad (\mathsf{si\_tcb\_id}, \mathsf{tcb\_cspace\_slot}) \mapsto_{\mathsf{cap}} cnode\_cap \wedge^* \\
&\quad (cnode\_id, dest\_root\_slot) \mapsto_{\mathsf{cap}} dest\_root\_cap \wedge^* \\
&\quad (cnode\_id, cnode\_cap\_slot) \mapsto_{\mathsf{cap}} cnode\_cap' \wedge^* \\
&\quad (cnode\_id, src\_slot) \mapsto_{\mathsf{cap}} src\_cap \wedge^* \\
&\quad (dest\_id, dest\_slot) \mapsto_{\mathsf{cap}} \mathsf{NullCap} \wedge^* R \gg s \wedge \\
&\quad \mathsf{one\_lvl\_lookup}\ cnode\_cap\ 32\ root\_size \wedge \\
&\quad \mathsf{one\_lvl\_lookup}\ cnode\_cap'\ src\_depth\ root\_size \wedge \\
&\quad \mathsf{one\_lvl\_lookup}\ dest\_root\_cap\ dest\_depth\ dest\_size \wedge \\
&\quad 0 < src\_depth \wedge src\_depth \leq 32 \wedge 0 < dest\_depth \wedge dest\_depth \leq 32 \wedge \\
&\quad \mathsf{is\_tcb}\ tcb \wedge \mathsf{is\_cnode\_cap}\ dest\_root\_cap \wedge \mathsf{is\_cnode\_cap}\ cnode\_cap \wedge \\
&\quad \mathsf{is\_cnode\_cap}\ cnode\_cap' \wedge src\_cap \neq \mathsf{NullCap} \wedge \\
&\quad \mathsf{guard\_equal}\ cnode\_cap\ src\_root\ 32 \wedge \mathsf{guard\_equal}\ cnode\_cap\ dest\_root\ 32 \wedge \\
&\quad \mathsf{guard\_equal}\ cnode\_cap\ src\_index\ src\_depth \wedge \mathsf{cap\_object}\ cnode\_cap = cnode\_id \wedge \\
&\quad \mathsf{guard\_equal}\ cnode\_cap'\ src\_index\ src\_depth \wedge \mathsf{cap\_object}\ cnode\_cap' = cnode\_id \wedge \\
&\quad \mathsf{guard\_equal}\ dest\_root\_cap\ dest\_index\ dest\_depth \wedge \mathsf{cap\_object}\ dest\_root\_cap = dest\_id \wedge \\
&\quad \mathsf{offset}\ src\_index\ root\_size = src\_slot \wedge \mathsf{offset}\ src\_root\ root\_size = cnode\_cap\_slot \wedge \\
&\quad \mathsf{offset}\ dest\_index\ dest\_size = dest\_slot \wedge \mathsf{offset}\ dest\_root\ root\_size = dest\_root\_slot\}
\end{aligned}
$$

$$
\mathsf{seL4\_CNode\_Move}\ dest\_root\ dest\_index\ dest\_depth\ src\_root\ src\_index\ src\_depth
$$

$$
\begin{aligned}
&\{\lambda s.\ \ll \mathsf{si\_tcb\_id} \mapsto_{\mathsf{fields}} tcb \wedge^* \\
&\quad cnode\_id \mapsto_{\mathsf{fields}} \mathsf{CNode}\ (\mathsf{empty\_cnode}\ root\_size) \wedge^* \\
&\quad dest\_id \mapsto_{\mathsf{fields}} \mathsf{CNode}\ (\mathsf{empty\_cnode}\ dest\_size) \wedge^* \\
&\quad (\mathsf{si\_tcb\_id}, \mathsf{tcb\_cspace\_slot}) \mapsto_{\mathsf{cap}} cnode\_cap \wedge^* \\
&\quad (\mathsf{si\_tcb\_id}, \mathsf{tcb\_pending\_op\_slot}) \mapsto_{\mathsf{cap}} \mathsf{RunningCap} \wedge^* \\
&\quad (cnode\_id, dest\_root\_slot) \mapsto_{\mathsf{cap}} dest\_root\_cap \wedge^* \\
&\quad (cnode\_id, cnode\_cap\_slot) \mapsto_{\mathsf{cap}} cnode\_cap' \wedge^* \\
&\quad (cnode\_id, src\_slot) \mapsto_{\mathsf{cap}} \mathsf{NullCap} \wedge^* \\
&\quad (dest\_id, dest\_slot) \mapsto_{\mathsf{cap}} src\_cap \wedge^* R \gg s\}
\end{aligned}
$$

Figure 7.3: API specification of seL4_CNode_Move

make kernel calls (and hence, only one thread can affect the system state). This allows the initialiser model to treat the kernel as a library that embeds the entire kernel state inside of the initialiser's state. It also allows us to avoid reasoning about interleaved user executions. To do so, we assume that the seL4 scheduler always schedules the initialiser. This works for our one-thread initialiser, but obviously would have to be generalised for more complex systems. This simpli-

fication is the reason why the formalisations for the seL4 system calls such as seL4_CNode_Move assume that the thread calling a kernel operation is the system initialiser (si_tcb_id).

7.5 Well-formed constraints and assumptions of the capDL specification

The system initialiser requires the capDL specifications that it initialises to be well-formed. We require this as capDL allows the specification of infeasible systems—for example, capDL allows objects to store an infinite number of capabilities and capabilities to point to an object of the wrong type.

These fundamental constraints are summarised as follows:

- There is only a finite number of objects in the system.
- Every object is of the correct size, with the correct number of capability slots.
- Every capability that points to an object points to a real object (for example, not an IRQ Node[9]), and there is a capability in the system for every real object. The types of the object and corresponding capability need to match.
- Each object only possesses capabilities of the right type, for example, page tables only store frame capabilities, whereas CNodes can contain most capability types.
- Capability rights are well formed, for example, frames cannot have write permissions without read permissions.
- Each capability has a unique original capability that it is derived from.
- Original capabilities must be in a default state and derived capabilities must be of a type that can be derived (for example, IRQ Handler capabilities cannot be derived).
- Page tables cannot be shared.
- Page tables must be empty or be mapped in a page directory.
- IRQ Nodes that are bound to an asynchronous endpoint must have an IRQ Handler capability pointing to them.
- The IRQ table cannot have two entries pointing to the same IRQ Node.

[9]IRQ Handler capabilities point to an IRQ number, which the IRQ table maps to an IRQ Node, rather than the IRQ Handler capability pointing directly to an IRQ Node.

There are further constraints in well_formed that encode current limitations of the initialiser, not fundamental constraints—we currently do not allow the system configuration to mention untyped capabilities, ASID pool capabilities, or the ASID and IRQ control capabilities that can be used to create new ASID pools and IRQ Handler capabilities respectively. This corresponds to static system configurations as used in a separation-kernel setting (Murray et al., 2013). Additionally, we do not support the domain capabilities nor the setting of the domain of a thread as these are recent additions to seL4.

These excluded capabilities break the assumption that there is a one-to-one correspondence between real objects and the original capability pointing to this object. This assumption is already relaxed with IRQ Handler capabilities (which do not point to an object, but an IRQ Number) and so, with the basic reasoning framework set up, we think these limitations can be lifted in future work.

Finally, our formal algorithm for system initialisation does not support mapping shared frames. Recall that our algorithm for system initialisation hands out all the capabilities it creates (except the capabilities it duplicates for the TCBs and CNodes). In seL4, the capability used to map a frame can also be used to unmap the frame, so if we share frames between two threads and wish to hand out the two capabilities used for each mapping, it is desirable to know where these capabilities used to map the frames should be placed. Unfortunately, the capDL specification language does not allow us to specify which capability is used to map a frame in an address space, which is why the initialiser does not support shared frames.[10] If the tracking of which capability was used to map a frame was added to the capDL specification language and the capDL kernel model, then this limitation in the initialiser could be addressed.[11] Our C implementation does support shared frames by not handing out the capabilities used to do the mappings and thus supports systems built with CAmkES (Kuz et al., 2007). This approach of not handing out the capabilities used to map frames could also be taken by our formal algorithm.

7.6 Decomposition of the final theorem

The key to this proof is the ability to decompose it along the functionality of the initialiser. In this section we explain how we do this decomposition and drill

[10] It is possible to give two threads a shared frame capability, allowing them both to map the shared frame themselves, which, while less than ideal, is a possible workaround.

[11] Technically this limitation is not apparent in our formal model (and is not encoded in well_formed) as the capDL kernel model non-deterministically allows a frame capability to be used to map a frame (due to the fact that it does not record if a capability has been used to map a frame).

into the proof of the creation of IRQ Handler capabilities and the handing out of capabilities as representative examples.

There are two aspects to this decomposition—decomposing the proof itself along function boundaries and decomposing predicates about objects, such as the predicate object_initialised, into smaller predicates. The former is provided by the frame rule (see Section 6.7), and the latter by our heap structure (see Section 6.5 and Section 6.6).

The proof of the system initialiser is divided into the same four sections that we divided the system initialiser into in Section 5.3.

- The first part ensures that parsing the kernel provided *bootinfo* structure correctly extracts the information about untyped memory and free capability slots in the boot state.
- The second part of the proof ensures that creation correctly occurs. It firstly ensures that the create_objects operation creates all the "real" objects described by the specification in their default state and stores the corresponding capabilities in the slots that later parts of the initialiser expect. This involves some internal book-keeping using the mapping *orig_caps* and looping over the collection of untyped capabilities. The second half of this part of the proof ensures that the create_irq_caps operation does similarly, that is, creating the required IRQ Handler capabilities, storing them in the correct slots, and storing the locations in the mapping *irq_caps*.
- The third part of the proof of system initialisation ensures that the original capabilities pointing to the CNodes and TCBs are duplicated and stored in the correct slots, as encoded in the mapping *dup_caps*.
- In the last, most complex part of the proof, we show that each object is transformed from a default state (described by the predicate object_empty) to its fully initialised state (described by the predicate object_initialised).

In the rest of this section, we examine the creation of IRQ Handler capabilities and the initialisation of capability spaces as representative examples of the structure of these proofs.

Each of these proofs maps a monadic function over a list. For example we map init_cnode over the list of CNodes, and similarly, we map init_cnode_slot over each capability slot inside a given CNode. To be able to reason about such maps, recall that we can reason about mapping a monadic function f over a distinct list xs using a rule such as Lemma 7.1. This rule is equivalent to the one presented in Section 6.4, except that it works on lifted heaps. This rule allows us to reason about how f transforms the state, such that the predicate P is true for each element

of the list *xs*, into a state where the predicate *Q* is true for each element of the list *xs*. This rule is utilised in a number of these proofs.

Lemma 7.1 ▶ mapM *rule*

$$\frac{\text{distinct } xs \qquad \forall R\, x.\ x \in xs \longrightarrow \{\lambda s.\ «P\ x \wedge^* I \wedge^* R» s\}\ f\ x\ \{\lambda s.\ «Q\ x \wedge^* I \wedge^* R» s\}}{\{\lambda s.\ «\bigwedge^*_{x \in xs} P\ x \wedge^* I \wedge^* R» s\}\ \mathsf{mapM}\ f\ xs\ \{\lambda s.\ «\bigwedge^*_{x \in xs} Q\ x \wedge^* I \wedge^* R» s\}}$$

Proof of object and capability creation

In this subsection we dive into the creation of IRQ Handler capabilities as a representative example for capability creation. As we can recall from Section 7.1, we used the predicates object_empty and object_initialised to describe the state of an object when it was first created and when it was correctly initialised, respectively. Similarly, we use the predicates irq_empty and irq_initialised to describe the same results for IRQ Nodes. Recall that each of these predicates requires an injection φ, which maps from the object identifiers used in our capDL specification to the physical memory addresses where the corresponding objects are located in the running kernel. The creation of real objects and IRQ Handler capabilities each produce their own injections, φ_{real} and φ_{IRQ} respectively, with each covering their respective objects, which are later combined to produce our final injection φ. We explain in detail how we construct this injective mapping φ in the remainder of this section.

While the creation of real objects uses a while loop, as opposed to a monadic map, (and is thus the only place where we reason about loop invariants), we examine the creation of IRQ Handler capabilities as a more illuminating example of both capability creation and the flexibility of our separation logic.

To explain how we prove that the creation of IRQ Handler capabilities is done correctly, we take a bottom-up approach, explaining first how we create a single IRQ Handler capability, and then build this up into a larger statement about the initialisation of all of the IRQ Handler capabilities. Our final theorem about the initialisation of IRQ Handlers, Theorem 7.3, is designed such that it can be combined with our theorems for the initialisation of other object types (some of which are shown later in Figure 7.4) to prove our top-level theorem, Theorem 7.1.

Our rule for creating a single IRQ Handler capability states that, given a free capability pointer *free_cptr* and an IRQ number that points to an empty IRQ Node in the kernel's IRQ table, the function create_irq_cap creates an IRQ Handler capability at the location *free_cptr* as recorded in the mapping *irq_caps*. The precondition is written in a form that fits well with our definition of si_irq_nodes, whilst the postcondition is written in a form that fits with our other theorems (by using the predicate irq_empty).

◀ **Lemma 7.2** create_irq_cap

$$\begin{array}{l}
\{\lambda s.\ {\ll}(\mathsf{si_cnode_id}, \mathit{free_cptr}) \mapsto_{\mathsf{cap}} \mathsf{NullCap} \wedge^* \\
\quad \mathit{irq} \mapsto_{\mathsf{IRQ}} \mathit{kernel_irq_id} \wedge^* \mathit{kernel_irq_id} \mapsto_{\mathsf{obj}} \mathsf{empty_irq_node} \wedge^* \\
\quad \mathsf{si_objects} \wedge^* R{\gg}\ s \wedge \\
\quad \mathsf{well_formed}\ \mathit{spec} \wedge \mathit{irq} \in \mathsf{used_irqs}\ \mathit{spec} \wedge \\
\quad \varphi_{IRQ}\ (\mathsf{cdl_irq_node}\ \mathit{spec}\ \mathit{irq}) = \mathsf{Some}\ \mathit{kernel_irq_id} \wedge \\
\quad \mathit{irq_caps}\ \mathit{irq} = \mathsf{Some}\ \mathit{free_cptr} \wedge \mathit{free_cptr} < 2^{\mathsf{si_cnode_size}}\} \\
\mathsf{create_irq_cap}\ \mathit{spec}\ (\mathit{irq}, \mathit{free_cptr}) \\
\{\lambda s.\ {\ll}\mathsf{irq_empty}\ \mathit{spec}\ \varphi_{IRQ}\ \mathit{irq} \wedge^* \\
\quad \mathsf{si_irq_cap_at}\ \mathit{irq_caps}\ \mathit{spec}\ \mathit{irq} \wedge^* \mathsf{si_objects} \wedge^* R{\gg}\ s\}
\end{array}$$

Proof: To prove this we simply expand all of the definitions of create_irq_cap, irq_empty, si_irq_cap_at, and si_objects and use the seL4 API specification for seL4_IRQControl_Get. ■

Moving up the stack, to prove create_irq_caps correctly initialises all of the required IRQ Handler capabilities, we need to provide the injection φ_{IRQ} that Lemma 7.2 requires. As opposed to the injection φ_{real}, constructed when creating the real objects capturing the subtlety that the kernel decides physical memory addresses at runtime, the injection φ_{IRQ} can be specified explicitly. We do this by exploiting the fact that a separation logic creates an algebraic structure that can be used to construct an injection, which we use to construct our injection φ_{IRQ}.

◀ **Lemma 7.3** sep_map_obj *injection*

$$\frac{\mathsf{finite}\ A \qquad (\bigwedge^*_{x \in A}\ (f\ x) \mapsto_{\mathsf{obj}} -)\ s}{\mathsf{inj\_on}\ f\ A}$$

Proof: This is true because it is not possible for two objects to be separately located at the same address. ■

Using this, we show an equivalent definition of si_irq_nodes to the one shown in Definition 7.6 that shows that the IRQ table of the kernel state must be an injection.

◀ **Lemma 7.4** si_irq_nodes *alternate definition*

$$\begin{array}{l}
\mathsf{si_irq_nodes}\ \mathit{spec}\ s = \\
(\exists \mathit{k_irq_table}.\ \mathsf{inj_on}\ \mathit{k_irq_table}\ (\mathsf{used_irqs}\ \mathit{spec}) \wedge \\
\quad (\bigwedge^*_{\mathit{irq} \in \mathsf{used_irqs}\ \mathit{spec}}\ (\mathit{irq} \mapsto_{\mathsf{IRQ}} \mathit{k_irq_table}\ \mathit{irq} \wedge^* \\
\qquad\qquad \mathit{k_irq_table}\ \mathit{irq} \mapsto_{\mathsf{obj}} \mathsf{empty_irq_node}))\ s)
\end{array}$$

We now combine the injections of the IRQ table of the kernel state and of the capDL specification to produce our injection φ_{IRQ} in the following rule. This rule states that, if that state contains |used_irqs *spec*| free capability pointers, the original capabilities as described by the *orig_caps* data structure, and the IRQ Table and Nodes (as encoded by si_irq_nodes), before executing create_irq_caps *spec free_cptrs*, then after executing the state will contain all the empty IRQ Nodes (as encoded by irq_empty), the IRQ Handler capabilities as described by *irq_caps*, and the original capabilities as described by *orig_caps*. The operation create_irq_caps returns a tuple containing the data structure *irq_caps* and the list of capability pointers that are still free (*free_cptrs_new*).

Lemma 7.5 ► create_irq_caps *helper*

$$
\begin{aligned}
&\{\lambda s.\ {\ll}\bigwedge^*_{cptr \in \mathsf{take}\ |\mathsf{used\_irqs}\ spec|\ free\_cptrs} (\mathsf{si\_cnode\_id}, cptr) \mapsto_{\mathsf{cap}} \mathsf{NullCap} \wedge^* \\
&\quad \bigwedge^*_{obj\_id \in \mathsf{real\_objects\_of}\ spec} \mathsf{si\_cap\_at}\ \varphi_{real}\ orig\_caps\ spec\ obj\_id \wedge^* \\
&\quad \mathsf{si\_objects} \wedge^* \mathsf{si\_irq\_nodes}\ spec \wedge^* R{\gg}\ s \wedge \\
&\quad \mathsf{well\_formed}\ spec \wedge |\mathsf{used\_irqs}\ spec| \leq |free\_cptrs| \wedge \\
&\quad \mathsf{distinct}\ free\_cptrs \wedge \bigwedge_{cptr \in free\_cptrs} cptr < 2^{\mathsf{si\_cnode\_size}}\} \\
&\mathsf{create\_irq\_caps}\ spec\ free\_cptrs \\
&\{\lambda(irq\_caps, free\_cptrs\_new)\ s. \\
&\quad \exists \varphi_{IRQ}.\ {\ll}\bigwedge^*_{irq \in \mathsf{used\_irqs}\ spec} \mathsf{irq\_empty}\ spec\ \varphi_{IRQ}\ irq \wedge^* \\
&\qquad \bigwedge^*_{irq \in \mathsf{used\_irqs}\ spec} \mathsf{si\_irq\_cap\_at}\ irq\_caps\ spec\ irq \wedge^* \\
&\qquad \bigwedge^*_{obj\_id \in \mathsf{real\_objects\_of}\ spec} \mathsf{si\_cap\_at}\ \varphi_{real}\ orig\_caps\ spec\ obj\_id \wedge^* \\
&\qquad \mathsf{si\_objects} \wedge^* R{\gg}\ s \wedge \\
&\quad irq\_caps = \mathsf{map\_of}\ (\mathsf{zip}\ (\mathsf{used\_irqs}\ spec)\ free\_cptrs) \wedge \\
&\quad free\_cptrs\_new = \mathsf{drop}\ |\mathsf{used\_irqs}\ spec|\ free\_cptrs \wedge \\
&\quad \mathsf{inj\_on}\ \varphi_{IRQ}\ (\mathsf{used\_irq\_cnodes}\ spec) \wedge \mathsf{dom}\ \varphi_{IRQ} = \mathsf{used\_irq\_cnodes}\ spec\}
\end{aligned}
$$

Proof: Firstly, the mapping φ_{IRQ} takes us from the identifiers for the IRQ Nodes in our capDL specification *spec* to the physical addresses of the corresponding IRQ Nodes in the running kernel. Since we know from our well_formed constraints of our capDL specification (outlined in Section 7.5) that the IRQ table of the capDL specification is an injection, and we know from Lemma 7.4 that the IRQ table of the running kernel must also be an injection, we compose the kernel's IRQ table with the inverse of the capDL specification's IRQ table to produce our injection φ_{IRQ}. The rest of the proof then follows by using our mapM rule (Lemma 7.1) and our rule for create_irq_cap proven (Lemma 7.2). ■

However, in this rule, we have two injections, φ_{real} and φ_{IRQ}, which cover different object ranges. To be able to combine them, we first show that they are indeed distinct, that if they are distinct we can combine them, and then show that we can rewrite our predicates to work on an extended mapping.

$$\frac{\begin{array}{c}\ll\bigwedge^*_{obj\_id\in\text{real\_objects\_of }spec}\ \text{object\_empty}\ spec\ \varphi_{real}\ obj\_id\ \wedge^*\\ \bigwedge^*_{irq\in\text{used\_irqs }spec}\ \text{irq\_empty}\ spec\ \varphi_{IRQ}\ irq\ \wedge^*\ R\gg s\\ \text{well\_formed}\ spec \qquad \text{inj\_on}\ \varphi_{real}\ (\text{real\_objects\_of}\ spec)\\ \text{inj\_on}\ \varphi_{IRQ}\ (\text{cdl\_irq\_node}\ spec \setminus \text{used\_irqs}\ spec)\\ \text{dom}\ \varphi_{real} = \text{real\_objects\_of}\ spec \qquad \text{dom}\ \varphi_{IRQ} = \text{cdl\_irq\_node}\ spec \setminus \text{used\_irqs}\ spec\end{array}}{\text{ran}\ \varphi_{real} \cap \text{ran}\ \varphi_{IRQ} = \emptyset}$$

◀ **Lemma 7.6** *Ranges distinct*

Proof: This is true because both of the predicates object_empty and irq_empty specify that an object exists for every real object identifier and IRQ number respectively, and since there cannot be two objects at the one physical memory address the ranges of the mappings must be disjoint. ■

Given then that the ranges of the two mappings are disjoint (from Lemma 7.6) and the domains of the two mappings are disjoint (as the real objects and IRQ Nodes are disjoint), we can combine the two mappings to produce a new injection using the following rule.

$$\frac{\begin{array}{cc}\text{inj\_on}\ m_L\ (\text{dom}\ m_L) & \text{inj\_on}\ m_R\ (\text{dom}\ m_R)\\ m_L \perp m_R & \text{ran}\ m_L \cap \text{ran}\ m_R = \emptyset\end{array}}{\text{inj\_on}\ (m_L +\!\!+ m_R)\ (\text{dom}\ m_L \cup \text{dom}\ m_R)}$$

◀ **Lemma 7.7** *Extending injections*

Now, we use this new injection in our predicates object_empty, irq_empty, and si_cap_at as long as we do not change any of the entries that they reference.

$$\frac{\text{dom}\ \varphi_{real} = obj\_ids \qquad \varphi_{real} \perp \varphi_{IRQ}}{\begin{array}{l}(\bigwedge^*_{obj\_id\in obj\_ids}\ \text{object\_empty}\ spec\ \varphi_{real}\ obj\_id) =\\ (\bigwedge^*_{obj\_id\in obj\_ids}\ \text{object\_empty}\ spec\ (\varphi_{real} +\!\!+ \varphi_{IRQ})\ obj\_id)\end{array}}$$

◀ **Lemma 7.8** *Extending* object_empty

$$\frac{\text{dom}\ \varphi_{IRQ} = \text{cdl\_irq\_node}\ spec \setminus irqs}{\begin{array}{l}(\bigwedge^*_{irq\in irqs}\ \text{irq\_empty}\ spec\ \varphi_{IRQ}\ irq) =\\ (\bigwedge^*_{irq\in irqs}\ \text{irq\_empty}\ spec\ (\varphi_{real} +\!\!+ \varphi_{IRQ})\ irq)\end{array}}$$

◀ **Lemma 7.9** *Extending* irq_empty

$$\frac{\text{dom}\ \varphi_{real} = obj\_ids \qquad \varphi_{real} \perp \varphi_{IRQ}}{\begin{array}{l}(\bigwedge^*_{obj\_id\in obj\_ids}\ \text{si\_cap\_at}\ \varphi_{real}\ si\_caps\ spec\ obj\_id) =\\ (\bigwedge^*_{obj\_id\in obj\_ids}\ \text{si\_cap\_at}\ (\varphi_{real} +\!\!+ \varphi_{IRQ})\ si\_caps\ spec\ obj\_id)\end{array}}$$

◀ **Lemma 7.10** *Extending* si_cap_at

Putting all of these lemmas together, we rewrite our earlier lemma about create_irq_caps, namely Lemma 7.5, into a form that can be combined with our

theorems about the creation and initialisation of the other object types to give our final theorem for create_irq_caps. This rule takes both the original list of free capability pointers that the system initialiser started with, namely *free_cptrs_orig*, and the list of free capability pointers that create_irq_caps uses, namely *free_cptrs*.

Theorem 7.3 ► create_irq_caps

$$
\begin{aligned}
&\{\lambda s.\ \exists \varphi_{real}.\\
&\quad \ll\!\bigwedge^*_{obj\_id \in \text{real\_objects\_of } spec} \text{object\_empty } spec\ \varphi_{real}\ obj\_id \wedge^*\\
&\quad \bigwedge^*_{obj\_id \in \text{real\_objects\_of } spec} \text{si\_cap\_at } \varphi_{real}\ orig\_caps\ spec\ obj\_id \wedge^* \text{si\_objects} \wedge^*\\
&\quad \text{si\_objects\_extra\_caps}'\ (\text{real\_objects\_of } spec)\ free\_cptrs\_orig\ ut\_cptrs \wedge^*\\
&\quad \text{si\_irq\_nodes } spec \wedge^* R\gg s \wedge\\
&\quad \text{well\_formed } spec \wedge |\text{used\_irqs } spec| \leq |free\_cptrs| \wedge\\
&\quad free\_cptrs = \text{drop } |\text{real\_objects\_of } spec|\ free\_cptrs\_orig \wedge\\
&\quad \text{distinct } free\_cptrs\_orig \wedge \bigwedge_{cptr \in free\_cptrs} cptr < 2^{\text{si\_cnode\_size}} \wedge\\
&\quad \text{inj\_on } \varphi_{real}\ (\text{real\_objects\_of } spec) \wedge \text{dom } \varphi_{real} = \text{real\_objects\_of } spec \wedge\\
&\quad \text{dom } orig\_caps = \text{real\_objects\_of } spec\}\\
&\text{create\_irq\_caps } spec\ free\_cptrs\\
&\{\lambda(irq\_caps, free\_cptrs\_new)\ s.\\
&\quad \exists \varphi.\ \ll\!\bigwedge^*_{obj\_id \in \text{real\_objects\_of } spec} \text{object\_empty } spec\ \varphi\ obj\_id \wedge^*\\
&\qquad \bigwedge^*_{irq \in \text{used\_irqs } spec} \text{irq\_empty } spec\ \varphi\ irq \wedge^*\\
&\qquad \bigwedge^*_{obj\_id \in \text{real\_objects\_of } spec} \text{si\_cap\_at } \varphi\ orig\_caps\ spec\ obj\_id \wedge^*\\
&\qquad \bigwedge^*_{irq \in \text{used\_irqs } spec} \text{si\_irq\_cap\_at } irq\_caps\ spec\ irq \wedge^* \text{si\_objects} \wedge^*\\
&\qquad \text{si\_objects\_extra\_caps}'\ (\text{objects\_of } spec)\ free\_cptrs\_orig\ ut\_cptrs \wedge^* R\gg s \wedge\\
&\quad irq\_caps = \text{map\_of } (\text{zip } (\text{used\_irqs } spec)\ free\_cptrs) \wedge\\
&\quad free\_cptrs\_new = \text{drop } |\text{used\_irqs } spec|\ free\_cptrs \wedge\\
&\quad \text{inj\_on } \varphi\ (\text{objects\_of } spec) \wedge \text{dom } \varphi = \text{objects\_of } spec\}
\end{aligned}
$$

Proof: To prove this, we use Lemma 7.5 and instantiate the existential injection φ with $\varphi_{real} +\!\!+ \varphi_{IRQ}$. We know that φ will be injective by Lemma 7.7, which requires that the ranges of φ_{real} and φ_{IRQ} are distinct, a fact we know from Lemma 7.6. Finally, we extend the predicates using Lemma 7.8, Lemma 7.9, and Lemma 7.10. ■

Object initialisation

Once all of the objects have been created, we then need to prove that we initialise each type of object correctly. We further divide this final part of the proof into separate proofs about the initialisation of each type of object. Recall that we categorised the different object types back in Section 5.3. Stateless objects are those with no state in capDL, that is, endpoints, asynchronous endpoints, and

frames. We separate the proof about the initialisation of the different types of real objects by showing the following rewrite rule.[12]

◀ **Theorem 7.4** *Object type decomposition*

$$\frac{\text{well\_formed } spec}{\begin{array}{l}(\bigwedge^*_{obj\_id \in \text{real\_objects\_of } spec} P\ obj\_id) = \\ (\bigwedge^*_{obj\_id \in \text{real\_cnodes\_of } spec} P\ obj\_id \wedge^* \bigwedge^*_{obj\_id \in \text{tcbs\_of } spec} P\ obj\_id \wedge^* \\ \bigwedge^*_{obj\_id \in \text{tables\_of } spec} P\ obj\_id \wedge^* \bigwedge^*_{obj\_id \in \text{stateless\_objects\_of } spec} P\ obj\_id)\end{array}}$$

Expanding this map of an arbitrary predicate over all real objects into maps by type allows us to use the frame condition in each rule to allow us to look at the initialisation of each type of object in isolation. As an example, consider the rules in Figure 7.4 for init_tcbs and init_cspace. We will go into the rule for init_cspace in detail in the remainder of this section, however, we can note here that each of these rules talk about a separate part of the overall object map, mention some side conditions about the presence of capabilities in the initialiser itself (si_cap_at φ *caps spec obj_id* and si_objects), and have a frame condition R than can be suitably instantiated to join them up to each other.

CNode initialisation

The proof of CNode initialisation is representative of the proofs of other object types. Capability slots in CNodes are initialised in a two-step process as described in Section 5.3. Each one of the CNodes has capabilities that are not original capabilities first *copied* in (each copy is copied from an original). Then, each CNode has the original capabilities *moved* in. We define a predicate cnode_half_initialised, shown in Definition 7.12, to describe this intermediate state of a CNode, when the non-original capabilities have been copied in. To do this, we first define the function cnode_half that takes a CNode, and returns a CNode with the original capabilities removed.

◀ **Definition 7.11** cnode_half

```
cnode_half spec obj_id obj ≡
 update_slots (λslot. if is_orig_cap (obj_id, slot) spec ∧ object_slots obj slot ≠ None
                      then Some NullCap
                      else object_slots obj slot) obj
```

Using this function, we define our predicate cnode_half_initialised in a similar way to how we defined the predicates for object_empty and object_initialised in Section 7.1.

[12] Recall from Section 5.3 that the capDL kernel model represents IRQ Nodes as CNodes of size zero, and so we use the term *real_cnodes spec* to describe the "real" CNodes (that is, not IRQ Nodes).

Theorem 7.5 ▶ init_tcbs

$$\begin{aligned}
&\{\lambda s.\ \ll\!\bigwedge^*_{obj_id \in \text{tcbs_of } spec} \text{object_empty } spec\ \varphi\ obj_id \wedge^* \\
&\quad \bigwedge^*_{obj_id \in \text{real_objects_of } spec} \text{si_cap_at } \varphi\ orig_caps\ spec\ obj_id \wedge^* \text{si_objects} \wedge^* R \gg s \wedge \\
&\quad \text{well_formed } spec \wedge obj_ids = \text{objects_of } spec \wedge \text{distinct } obj_ids\} \\
&\text{init_tcbs } spec\ orig_caps\ obj_ids \\
&\{\lambda s.\ \ll\!\bigwedge^*_{obj_id \in \text{tcbs_of } spec} \text{object_initialised } spec\ \varphi\ obj_id \wedge^* \\
&\quad \bigwedge^*_{obj_id \in \text{real_objects_of } spec} \text{si_cap_at } \varphi\ orig_caps\ spec\ obj_id \wedge^* \text{si_objects} \wedge^* R \gg s\}
\end{aligned}$$

Theorem 7.6 ▶ init_cspace

$$\begin{aligned}
&\{\lambda s.\ \ll\!\bigwedge^*_{obj_id \in \text{real_cnodes_of } spec} \text{object_empty } spec\ \varphi\ obj_id \wedge^* \\
&\quad \bigwedge^*_{obj_id \in \text{real_objects_of } spec} \text{si_cap_at } \varphi\ orig_caps\ spec\ obj_id \wedge^* \\
&\quad \bigwedge^*_{irq \in \text{used_irqs } spec} \text{si_irq_cap_at } irq_caps\ spec\ irq \wedge^* \\
&\quad \bigwedge^*_{obj_id \in \text{real_cnodes_and_tcbs_of } spec} \text{si_cap_at } \varphi\ dup_caps\ spec\ obj_id \wedge^* \\
&\quad \text{si_objects} \wedge^* R \gg s \wedge \\
&\quad \text{well_formed } spec \wedge obj_ids = \text{objects_of } spec \wedge |obj_ids| \leq |free_cptrs| \wedge \\
&\quad \text{distinct } obj_ids \wedge \text{distinct } free_cptrs \wedge \\
&\quad orig_caps = \text{map_of } (\text{zip } [obj \leftarrow obj_ids.\ \text{real_object_at } obj\ spec]\ free_cptrs) \wedge \\
&\quad irq_caps = \text{map_of } (\text{zip } (\text{used_irqs } spec)\ (\text{drop } |\text{real_objects_of } spec|\ free_cptrs))\} \\
&\text{init_cspace } spec\ orig_caps\ dup_caps\ irq_caps\ obj_ids \\
&\{\lambda s.\ \ll\!\bigwedge^*_{obj_id \in \text{real_cnodes_of } spec} \text{object_initialised } spec\ \varphi\ obj_id \wedge^* \\
&\quad \bigwedge^*_{cptr \in \text{take } |\text{objects_of } spec|\ free_cptrs} (\text{si_cnode_id}, cptr) \mapsto_{\text{cap}} \text{NullCap} \wedge^* \\
&\quad \bigwedge^*_{obj_id \in \text{real_cnodes_and_tcbs_of } spec} \text{si_cap_at } \varphi\ dup_caps\ spec\ obj_id \wedge^* \\
&\quad \text{si_objects} \wedge^* R \gg s\}
\end{aligned}$$

Figure 7.4: Individual rules for init_tcbs and init_cspace.

Definition 7.12 ▶ cnode_half_initialised

$$\begin{aligned}
&\text{cnode_half_initialised } spec\ \varphi\ obj_id\ s \equiv \\
&\exists kernel_obj_id\ spec_object.\ \varphi\ obj_id = \text{Some } kernel_obj_id \wedge \\
&\quad (kernel_obj_id \mapsto_{\text{obj}} (\text{spec2s } \varphi \circ \text{cnode_half } spec\ obj_id)\ spec_object)\ s \wedge \\
&\quad \text{cdl_objects } spec\ obj_id = \text{Some } spec_object
\end{aligned}$$

We use our rule for monadic maps mapM, Lemma 7.1, to prove both halves of the initialisation of all of the CNodes. We use it first to transform the proof of half-initialising *all* of the CNodes into a proof of half-initialising a *single* CNode. We then decompose the problem of initialising a single object into the separate parts of an object, namely its fields and its individual capability slots. This is embodied in Theorem 7.7.

$$\frac{\text{well\_formed } \mathit{spec}}{\begin{array}{l}\text{object\_initialised } \mathit{spec}\ \varphi\ \mathit{obj\_id} = \\ (\text{object\_fields\_initialised } \mathit{spec}\ \varphi\ \mathit{obj\_id} \wedge^* \\ \bigwedge^*_{\mathit{slot} \in \text{obj\_slots } \mathit{obj\_id}\ \mathit{spec}} \text{object\_slot\_initialised } \mathit{spec}\ \varphi\ \mathit{obj\_id}\ \mathit{slot} \wedge^* \\ \text{object\_empty\_slots\_initialised } \mathit{spec}\ \varphi\ \mathit{obj\_id})\end{array}}$$

◀ **Theorem 7.7** object_initialised *decomposition*

Such a decomposition is not necessarily true for any separation logic and any definition of partly initialised objects. Being able to prove the rule above as an equality was one of the design goals of our separation logic. In particular, the definition of object_initialised (see Section 7.3) contains an existential quantifier over *kernel_obj_id* and *spec_object* which needs to be well-behaved enough to lift over the separating conjunction on the right hand side of the rule.

We use Theorem 7.7 (and similar rules for decomposing object_empty and cnode_half_initialised) to decompose the proof of initialising a single CNode into a proof of initialising the fields of an object, the capability slots of an object, and the empty slots of an object. The empty slots of an object need no initialisation, the only field of a CNode is its size (which is set when the CNode is created), and so we are left with a proof of how to initialise *all* of the capability slots of a CNode. To prove all the capability slots we use Lemma 7.1 again to decompose this into a proof of initialising a *single* capability slot. The ability to be able to use the same rule for both loops shows some of the power of our separation logic.

Arriving at the leaf kernel calls of the init_cnode operation, the initialiser extracts the capabilities that authorise it to make these calls. The presence of these capabilities in the correct locations is encoded in the predicates such as si_cap_at, as introduced in Section 7.2. We use the rule shown in Lemma 7.11 to extract the specific capability we need from the precondition and postcondition of a lemma. We then join the initialiser proof with formal specifications of the seL4 API that were introduced in Section 7.4, expanding the definitions of object_slot_initialised and similar to complete this proof.

$$\frac{\text{finite } A \qquad x \in A \qquad \forall R.\ \{\lambda s.\ {\ll}P \wedge^* I\ x \wedge^* R{\gg}\ s\}\ f\ \{\lambda s.\ {\ll}Q \wedge^* I\ x \wedge^* R{\gg}\ s\}}{\{\lambda s.\ {\ll}P \wedge^* \bigwedge^*_{x' \in A} I\ x' \wedge^* R{\gg}\ s\}\ f\ \{\lambda s.\ {\ll}Q \wedge^* \bigwedge^*_{x' \in A} I\ x' \wedge^* R{\gg}\ s\}}$$

◀ **Lemma 7.11** *Separation conjunction invariant extraction*

Proof: To be able to use the Hoare triple in our assumption, we first break the separation logic predicate $\bigwedge^*_{x' \in A} I\ x'$ into $\bigwedge^*_{x' \in A - \{x\}} I\ x' \wedge^* I\ x$. Then, since our assumption requires $\{\lambda s.\ {\ll}P \wedge^* I\ x \wedge^* R{\gg}\ s\}\ f\ \{\lambda s.\ {\ll}Q \wedge^* I\ x \wedge^* R{\gg}\ s\}$ is true for *any* R, we can prove our conclusion by setting the R in our assumptions to $\bigwedge^*_{x' \in A - \{x\}} I\ x' \wedge^* R$. ■

Rewriting predicates

Unfortunately the proof above of init_cspace ignores one crucial fact. When reasoning about monadic maps (that is, loops constructed using mapM), our rules such as Lemma 7.1, require that the predicates P and Q be folded over the same list xs as the function f is mapped over. Recall from Section 5.3, the operation init_cspace is mapped first over each CNode, and then over each CNode slot (and this is done twice, first to copy non-original capabilities, and second to move the original capabilities).

Examining the lemma for init_cspace, while the predicates object_empty and object_initialised are defined using the set of CNodes that init_cspace loops over, the other predicates such as si_cap_at φ *orig_caps spec*, si_irq_cap_at *irq_caps spec*, and si_cap_at φ *dup_caps spec* are defined over all the real objects, the IRQ Nodes that are used in the specification, and the CNodes *and* TCBs of a specification respectively. To be able to use our mapM rules, such as Lemma 7.1, we need to rewrite each of these predicates into a form that is defined using the set of CNodes (and possibly also the capability slots inside these CNodes).

For this reason, to prove Theorem 7.6, we first prove the following rule. This version of the rule has all of the separation predicates defined using the same list that init_cspace loops over, and so fits well with our mapM rules and is the lemma we actually proved above.

Lemma 7.12 ▸ init_cspace *helper*

$$\begin{aligned}
&\{\lambda s.\ \ll\bigwedge^*_{obj_id\in \text{real_cnodes_of } spec} \text{object_empty } spec\ \varphi\ obj_id \wedge^* \\
&\quad \bigwedge^*_{obj_id\in \text{real_cnodes_of } spec} \text{si_obj_caps_at } \varphi\ orig_caps\ spec\ obj_id \wedge^* \\
&\quad \bigwedge^*_{obj_id\in \text{real_cnodes_of } spec} \text{si_spec_irq_caps_at } irq_caps\ spec\ obj_id \wedge^* \\
&\quad \bigwedge^*_{obj_id\in \text{real_cnodes_of } spec} \text{si_cap_at } \varphi\ dup_caps\ spec\ obj_id \wedge^* \text{si_objects} \wedge^* R\gg s \wedge \\
&\quad \text{well_formed } spec \wedge obj_ids = \text{objects_of } spec \wedge \text{distinct } obj_ids\} \\
&\text{init_cspace } spec\ orig_caps\ dup_caps\ irq_caps\ obj_ids \\
&\{\lambda s.\ \ll\bigwedge^*_{obj_id\in \text{real_cnodes_of } spec} \text{object_initialised } spec\ \varphi\ obj_id \wedge^* \\
&\quad \bigwedge^*_{obj_id\in \text{real_cnodes_of } spec} \text{si_spec_obj_null_caps_at } \varphi\ orig_caps\ spec\ obj_id \wedge^* \\
&\quad \bigwedge^*_{obj_id\in \text{real_cnodes_of } spec} \text{si_spec_irq_null_caps_at } irq_caps\ spec\ obj_id \wedge^* \\
&\quad \bigwedge^*_{obj_id\in \text{real_cnodes_of } spec} \text{si_cap_at } \varphi\ dup_caps\ spec\ obj_id \wedge^* \text{si_objects} \wedge^* R\gg s\}
\end{aligned}$$

In the rest of this section, we show how we use this helper lemma, Lemma 7.12, to prove our top-level theorem for init_cspace, Theorem 7.6. To do so, we concentrate on the conversion of the predicate about the capabilities pointed to by the mapping *orig_caps*, from being defined on the real objects of a capDL specification *spec* (that is, $\bigwedge^*_{obj\_id\in \text{real\_objects\_of } spec}$ si_cap_at φ *orig_caps spec obj_id*) into a predicate defined on the CNodes of a capDL specification *spec* (that is,

$\bigwedge^*_{obj\_id \in real\_cnodes\ spec}$ si_obj_caps_at φ *orig_caps spec obj_id*) as a representative example. We prove this conversion at the end of this section in Theorem 7.8.

The former predicate, defined using si_cap_at, specifies that for each real object there is a master capability pointing to it (in the specified capability slot in the system initialiser's capability space). Our new predicate, defined using si_obj_caps_at, specifies that the system initialiser contains all the original capabilities that are required by the CNodes that it is initialising. Of course, since there exists a one-to-one correspondence between original capabilities handed out by the initialiser and the real objects the initialiser creates, we are able to prove these two predicates are equivalent.

To define si_obj_caps_at, we first define the predicate si_obj_cap_at which specifies, for a particular capability slot, if the capability slot is one where an original capability is located, and the capability located there has an object (recall that IRQ Handler capabilities do not), then our predicate specifies that there must exist a capability *spec_cap* in the slot, and the predicate si_cap_at φ *si_caps spec* is true for the object that *spec_cap* points to.

Definition 7.13 si_obj_cap_at

si_obj_cap_at φ *si_caps spec obj_id slot* $\equiv$
if is_orig_cap $(obj\_id, slot)$ *spec* $\wedge$ cap_at cap_has_object $(obj\_id, slot)$ *spec*
then $\lambda s.\ \exists spec\_cap.$ si_cap_at φ *si_caps spec* (cap_object *spec_cap*) $s \wedge$
opt_cap $(obj\_id, slot)$ *spec* = Some *spec_cap*
else $\square$

Using our new predicate si_obj_cap_at, we define si_obj_caps_at which specifies that si_obj_cap_at is true for each capability slot of an object.

Definition 7.14 si_obj_caps_at

si_obj_caps_at φ *si_caps spec obj_id* $\equiv$
$\bigwedge^*_{slot \in \text{obj\_slots}\ obj\_id\ spec}$ si_obj_cap_at φ *si_caps spec obj_id slot*

As we can see, this new predicate is defined in such a way that, given a one-to-one correspondence between original capabilities and real objects, we can prove the two predicates equivalent. To prove this one-to-one correspondence between original capabilities and real objects, we define the function object_at_cap_slot, which, given a capability slot in a given CNode, returns the object identifier pointed to by the capability in a specific capability slot (assuming this exists, and otherwise the return result is undefined). We show in Lemma 7.13 that the function object_at_cap_slot is a bijection between the capability slots of a CNode containing original capabilities that point to objects, and the real objects in a system.

Lemma 7.13 ▸ *Original capabilities and real objects bijection*

$$\frac{\mathsf{well\_formed}\ spec}{\begin{array}{l}\mathsf{bij\_betw}\ (\mathsf{object\_at\_cap\_slot}\ spec)\\ \{(obj\_id, slot) \mid \mathsf{is\_orig\_cap}\ (obj\_id, slot)\ spec \wedge\\ \qquad\qquad \mathsf{cap\_at\_to\_real\_object}\ (obj\_id, slot)\ spec \wedge\\ \qquad\qquad \mathsf{real\_cnode\_at}\ obj\_id\ spec\}\\ (\mathsf{real\_objects\_of}\ spec)\end{array}}$$

Proof: The fact that this function is a bijection between original capabilities in real CNodes (that is, not IRQ Nodes) and real objects is due to the constraints we place on the capDL specification *spec* in the predicate well_formed *spec*, which mandates that this must be the case, namely, that all original caps must be unique (that is, two original capabilities cannot point to the same object), each real object must have an original capability pointing to it, and all capabilities pointing to real objects must point to an object. ■

To prove equivalence between our two predicates describing the original capabilities, firstly $\bigwedge^*_{obj\_id \in \mathsf{real\_objects\_of}\ spec}$ si_cap_at φ *orig_caps spec obj_id* (which maps over each of the real objects of a specification) and the second predicate $\bigwedge^*_{obj\_id \in real\_cnodes\ spec}$ si_obj_caps_at φ *orig_caps spec obj_id* (which maps over each of the capability slots of all of the CNodes of a specification), we need a way of rewriting a nested separation conjunction. Recall that, because separation conjunction is an associative, commutative operation, it is well defined to fold a separation predicate over a set. We rewrite a nesting of these folds using the following lemma, Lemma 7.14.

Lemma 7.14 ▸ *Nested separation conjunction*

$$\frac{\mathsf{finite}\ A \qquad \forall x \in A.\ \mathsf{finite}\ (B\ x)}{(\bigwedge^*_{x \in A} \bigwedge^*_{y \in B\ x} P\ x\ y) = (\bigwedge^*_{(x,y) \in (\bigcup_{x \in A} \bigcup_{y \in B\ x} \{(x,y)\})} P\ x\ y)}$$

Next, we need a way of transforming predicates defined on one set to another. We can transform the folding of a predicate P over one set A, to the folding of a predicate Q over another set B, providing we have an bijection f between the two sets A and B, and that f transforms between the two predicates in the obvious manner, as shown in Lemma 7.15.

Lemma 7.15 ▸ *Lifted separation conjunction reindexing*

$$\frac{\mathsf{bij\_betw}\ f\ A\ B \qquad \forall a.\ a \in A \longrightarrow P\ a = Q\ (f\ a)}{(\bigwedge^*_{x \in A} P\ x) = (\bigwedge^*_{x \in B} Q\ x)}$$

Using these results, we now rewrite our predicate si_cap_at, which specifies that there must be an original capability for each real object, into a predicate that

specifies that each capability slot with an original capability must point to a real object, as shown in Theorem 7.8.

$$\frac{\text{well\_formed } spec}{\begin{array}{l}(\bigwedge^*_{obj\_id \in \text{real\_objects\_of } spec} \text{si\_cap\_at } \varphi\ orig\_caps\ spec\ obj\_id) = \\ (\bigwedge^*_{obj\_id \in \text{real\_cnodes\_of } spec} \text{si\_obj\_caps\_at } \varphi\ orig\_caps\ spec\ obj\_id)\end{array}}$$

◀ **Theorem 7.8**
si_caps_at *conversion*

Proof: We first expand the definition of si_obj_caps_at and rewrite it using our lemma for nested separation conjunction, Lemma 7.14, to produce a new predicate, folded over a set of $(obj\_id, slot)$ tuples. We then rewrite this new predicate using Lemma 7.15, where we set the bijection f in the rule to be object_at_cap_slot *spec*. Finally, we know that object_at_cap_slot *spec* is a bijection from Lemma 7.13, completing the proof. ■

Using this rewrite, and similar one for the IRQ Handler capabilities, there is one remaining part left to be able to transform our theorem for init_cspace, shown in Theorem 7.6, into a form that can be used with our mapM rules, namely Lemma 7.12. This last remaining difference between the two rules is the inclusion of the duplicate capabilities for the TCBs in the former rule, but not the latter. These can be safely ignored and placed in the frame of the latter rule in a similar way to Lemma 7.11. With all these pieces in place, we have shown how we correctly initialise capability spaces, and show that all of the original capabilities are handed out by the system initialiser.

7.7 Conclusion

In this chapter, we have examined the proof of a formal model of the algorithm for the system initialiser with respect to a formalised API for the seL4 microkernel. By joining these proofs to a verified abstraction of seL4, we are forced to deal with the intricacies of a low-level kernel API in our proofs, the complexity of which is illustrated in Section 7.4.

We have shown that our separation logic is suitable to prove both these kernel API specifications, as well as the specification of our user-level program—the system initialiser. Designing our separation logic to have the right level of granularity and decomposability was crucial for reasoning about individual components of each object separately.

While the proof of functional correctness of seL4, and other proofs building on the functional correctness of seL4, had to show global invariants that are inconvenient to express in separation logic, we instead express the properties we needed, such as the original capabilities that the initialiser creates, and where they

are stored using separation logic predicates. Not requiring any invariants made the proof much more modular and easier to extend.

The system initialiser makes heavy use of nested loops. Again, our separation logic setup enabled us to decompose these loops into local steps without stating complex invariants.

We think that this model of using a separation logic for the specification of kernel APIs and for proving the initialisation of user-level systems is applicable in capability-based microkernels, as is using an abstract separation logic for a large-scale proof. Its modular, decomposable reasoning without invariants made it ideal for this style of proof.

Chapter Summary

- We defined precisely what it means for a system to be correctly initialised in conformance with a given capDL specification. In particular, every object in the specification must be present in the kernel state at some physical memory address.
- We defined the state of the system initialiser, the objects it contains and the capabilities it possesses throughout initialisation.
- We have decomposed our proof along the various stages of initialisation, initialising the objects type-by-type, object-by-object, and capability-by-capability.
- We have joined our proof to a formalised API for seL4 that was proven using the separation logic we defined in Chapter 6. By joining our proof to a verified abstraction of seL4, we are forced to deal with the intricacies of a low-level kernel API in our proofs.
- We have explored the creation of IRQ Handler capabilities and the initialisation of capability spaces as illustrative examples of the style of proof that we have done in this work. All of the proofs and definitions have been published under an open-source licence (Trustworthy Systems Team, 2014).
- By formally reasoning about the correctness of initialisation, we can have confidence that we can start a system correctly and that seL4 will enforce the security of our system as specified.

Chapter 8

Conclusion

8.1 Discussion

In this thesis we[1] presented the formalisation and correctness proof of a generic, automatic system initialiser that brings a seL4-based system from its initial boot state into a desired configuration. From such a configuration, we can then reason with confidence about the security of the resulting system.

We have examined how capability-based systems, based on the take-grant protection model, can give us guarantees about the authority confinement and possible information flow of the entire system solely through the examination of the capabilities.

This result predates the work of Sewell et al. (2011), Murray et al. (2013), and Boyton et al. (2013) which proved stronger results directly on the abstract model of the seL4 microkernel. Combined, these latter works show that we can reason about the access control, integrity, and information flow of this system using only the capabilities present in the capDL kernel model, and know that these results hold on the kernel implementation. This gives us great confidence that we can build systems using our formal algorithm of system initialisation that we presented here, and hence be able to have confidence in the overall system.

We have additionally shown a general separation logic framework that can be used to reason about such user-level systems, have produced a proof framework to reason about user-level executions on top of a formally verified microkernel API, and have applied it to show the correctness of the initialiser model.

There are of course a number of limitations in this work. The initialiser we present here is specific to the capability-based microkernel seL4. We think that the general principle and pattern of reasoning we have employed in this work would

[1] Throughout this thesis we have used the royal "we" to indicate the author.

generalise to other capability-based systems, although the feasibility of doing so has not been examined.

We have aimed to design a generic separation logic framework for reasoning about user-level executions on top of a formally verified microkernel API. So far this framework has only been used to show the correctness of the system initialiser *algorithm*. Formally connecting this to our C implementation remains a target for our future work.

The results we have proven here are all safety and not liveness results. That is, we prove that if the system initialiser successfully terminates, then it will have set up the system correctly. This is due to the non-deterministic nature of the capDL kernel model. We could gain confidence in the liveness of our algorithm through testing, code inspection, or informal reasoning. Alternatively, we could gain a guarantee of liveness by formally reasoning at a lower-level of abstraction, but this is outside the scope of this thesis.

We discuss a number of further limitations and how they could be addressed in future work in Section 8.3.

8.2 Implementation experience

It is worth noting some of the design decisions that we made and explore some of their implications. We discuss three such decisions below.

Separation logic We have found that designing a separation logic with the right level of granularity has been invaluable for developing the proofs in this thesis. We have found separation logic allows for convenient expression of both what an API *does* and what it leaves *unchanged*. Unfortunately the API specifications developed for (but outside the scope of) this thesis, as illustrated in Theorem 7.2, cannot be described as concise. We could attempt to simplify such theorems by creating definitions that wrap up many of the preconditions, although another problem remains if we wished to use these rules in other contexts, namely the need for some operations to require multiple rules to specify their behaviour, which we explain below.

The separation logic predicates we describe must be disjoint. For example, the predicate $obj\_id \mapsto_{\mathsf{obj}} obj \wedge^* obj\_id \mapsto_{\mathsf{obj}} obj$ is never true, as two predicates about the same object location cannot be disjoint.[2] What this means is that to specify the behaviour of an operation such as swap_objects, which swaps the objects at two locations, then Lemma 8.1 can only be used when the locations are different, that is, when $obj\_id \neq obj\_id'$.

[2] This is the same reasoning we used to prove that separation predicates create natural injections, as shown in Lemma 7.3.

$$\{obj\_id \mapsto_{\text{obj}} obj \wedge^* obj\_id' \mapsto_{\text{obj}} obj' \wedge^* R\}$$
$$\text{swap\_objects } obj\_id\ obj\_id'$$
$$\{obj\_id \mapsto_{\text{obj}} obj' \wedge^* obj\_id' \mapsto_{\text{obj}} obj \wedge^* R\}$$

◀ **Lemma 8.1** *Swap object (different object identifiers)*

To reason that calling swap_objects *obj_id obj_id* leaves the object located at *obj_id* unchanged, we need a different rule, namely Lemma 8.2.

$$\{obj\_id \mapsto_{\text{obj}} obj \wedge^* R\}$$
$$\text{swap\_objects } obj\_id\ obj\_id$$
$$\{obj\_id \mapsto_{\text{obj}} obj \wedge^* R\}$$

◀ **Lemma 8.2** *Swap object (same object identifier)*

When moving capabilities in seL4 using seL4_CNode_Move, there are 4 possible scenarios, each of which requires a custom rule—the capability being moved could be in the root task's CNode, or another CNode the root task has access to, and the location that the capability is being moved to could be in the root task's CNode, or another CNode the root task has access to. Thankfully, in this work, we only required one version of this rule, namely Theorem 7.2, but this situation is less than ideal.

A separation logic with permissions, such as that of Bornat et al. (2005), Parkinson (2005), or Dockins et al. (2009), allows reasoning about concurrent programs where multiple operations concurrently read the same data as long as no operation is simultaneously writing to the data. Whilst using such a separation logic would not help us when proving rules for swapping objects, it could help us when reasoning about moving capabilities using seL4_CNode_Move, due to the fine level of granularity of our predicates.[3] A separation logic with permissions comes with greater cost of complexity to the separation logic itself and the reasoning about it, but it might reduce the number of rules required. The tradeoffs between such systems are not addressed here.

Local specifications Separation logic requires our specification to be local, that is, operations of our specification cannot depend on, or modify, the whole program state. We found that while the operations in the capDL kernel model (which were not originally designed with separation logic in mind) were often local in their behaviour, many parts were not. We found that there was sometimes a tension between elegance and locality when writing a monadic specification—it was

[3] A separation logic with permissions generally allows multiple readers, but only one writer, which is why it does not help with swap_objects, but the collisions in our rule for seL4_CNode_Move come from reading the various CNode's fields, not writing their capability slots, hence a separation logic with permissions may help.

common in many parts of the specification to state "for all objects in the specification, non-deterministically pick one and do something to it". For example, should the fact that a thread is schedulable be stored in the thread itself or in a global schedulable set? As capDL is an abstraction of the behaviour of the C implementation of seL4 we are free to do either, but one model allows separation logic to reason about scheduling and one does not.

While many parts of the capDL kernel model were changed to allow a separation logic specification for their operation, other non-local operations such as scheduling remain. This non-locality of scheduling in capDL precludes us from reasoning about scheduling using separation logic. This could be addressed in future work.

Proof size We found that using a separation logic for reasoning required custom tactics to be constructed to be able to manage the large predicates used. The tactics that were developed along with the proof and earlier proofs could be greatly simplified using the more advanced tactics for weakest-precondition style reasoning.

Overall, the formal algorithm for the system initialiser is around 500 lines, whereas the proof of the system initialiser algorithm is around 12,000 lines. The connection to the fully realistic kernel model is the main source of complexity in the proof. In comparison, there are 7,600 lines for the proof of the capDL kernel API specification, 27,000 lines for the refinement proof between the capDL kernel model and the functional specification, and 200,000 lines for the functional specification to the C code of the kernel (Klein et al., 2009).[4] The take-grant protection model we presented in Chapter 4 is much smaller, at around 1,700 lines of specification and proof.

These specifications and proofs are all available as part of the seL4 release (Trustworthy Systems Team, 2014). The C implementation of the initialiser has also been released (Boyton et al., 2014).

8.3 Future work

There are a number of limitations and areas of future work that could be addressed.

The final result of this thesis, that the system has been started in conformance with a given capDL specification has not been formally linked to the access control policy of Sewell et al. (2011) and Murray et al. (2013). The capDL kernel model

[4] In absence of good metrics for counting the size of a proof, we use `wc -l` which simply counts the number of lines in each file, including all comments and blank lines.

itself has been linked to these access control policy, as shown in the work of Boyton et al. (2013), but there are a few pieces missing before the work of this thesis can be joined up to these access control policies. Firstly, our proof statement is a separation logic statement, whereas the access control policies are specified in traditional Hoare logic. We can convert from a separation logic specification of the state to a more traditional form, but this uncovers the problem of identifiers. Throughout this work we have assumed capDL identifiers are simple names, not memory addresses as they are assumed to be in the link between capDL and the access control policies. This is correspondence between the two is encoded in the injection φ in our final statement proof statement, Theorem 7.1.

As the work of Boyton et al. (2013) which links capDL to the access control policies assumes that capDL identifiers correspond to the memory addresses in an access control policy, this mismatch could be addressed by changing this link between capDL and the access control policies, by changing the state relation of the capDL refinement, or by removing the implicit assumptions about the physical memory addresses in the proofs of Sewell et al. (2011) and Murray et al. (2013). We leave reconciling these to future work.

Limitations of our proof

There are also a number of limitations in the kind of capDL specifications that we initialise, as detailed in Section 7.5. The main limitations are the exclusion of various capability types (untyped capabilities, ASID pool capabilities, and the ASID and IRQ control capabilities), and the lack of support for shared frames.

The capabilities that we exclude break the one-to-one correspondence between objects and the original capability pointing to this object. As we have lifted this assumption to support IRQ Handler capabilities, we believe we could extend the algorithm and the proof to support these additional capabilities in a similar way.

seL4 requires a new frame capability to be used for each frame-mapping in an address space. A thread holding the frame capability that was used to map a frame possesses the authority to unmap that frame. At present, the capDL kernel specification does not track this correspondence, and so we cannot use capDL to safely specify a system with shared frames, lest two threads accidentally be given the frame capability used to map the other's frame. If capDL was extended to track this mapping, then the proofs could be updated to allow shared frames.

The kernel interaction model developed for this thesis embeds the kernel state inside that of the initialiser and treats the kernel as an API, as explained in Section 7.4. While this is how developers often view an operating system, it is not how this operating system actually works. Flipping this around so the kernel is the

one scheduling threads is outside the scope of this thesis. This modelling of the seL4 interaction also assumes that the libseL4 API (the API used by a developer, such as `seL4_CNode_Move`) corresponds to the intents of the capDL kernel model, which has been axiomatised in our work. Proving this would require reasoning about the kernel/user-space boundary and the assembly code system-call stubs.

At present, we wipe the intent and ASIDs of memory objects when we lift the heap; it would be nice to avoid both. The former could be addressed by changes to the intent modelling in capDL to make the setting of intents a local operation[5], and the latter could be changed by more formal reasoning about the ASIDs of memory objects. As the ASID of a memory object is determined by the kernel at runtime, this would most likely require constructing another injection, similar to the one mapping from object identifiers of a capDL specification to the physical memory addresses where corresponding objects are located.

Finally, we do not reason about the behaviour of the scheduler, as the capDL kernel model does not model the scheduling of threads with a sufficient level of detail. For this reason, we have assumed that the root task is the only task that runs, and have not reasoned about starting threads.

Implementation refinement

While care has been taken to implement the formal algorithm of system initialisation in C, we could greatly increase our confidence in the implementation by formally proving refinement between the C code and our abstract algorithm. Generally, such refinement proofs are rather labour intensive, especially for codebases such as the seL4 microkernel.

We expect the refinement of the initialiser code to be much simpler than that of the seL4 kernel for two main reasons. Firstly, the system initialiser code is much simpler than the code of seL4, as the initialiser code consists mostly of simple loops—its complexity lies in the interaction with the kernel, which we have treated in the proof of the correctness of the algorithm. Secondly, there have been great recent advances in tools for proving refinement of C code, such as AutoCorres (Greenaway et al., 2012; Greenaway et al., 2014). Our C code has been specifically written in such a way that it can be used by such tools.

Turn the system initialiser into a library for clients to use

The ability to create and configure objects in conformance with a specification is useful not just for the root task of an operating system (that is, the very first

[5]To make setting a thread's intent a local operation, intents would need to be (at least partially) stored outside the TCB in a data structure which would be shared by TCBs that share an IPC buffer. This change would also more closely model reality.

task that runs in the system), but for other processes in a system, which may also wish to dynamically setup new subsystems at runtime. The root task that we created that does system initialisation could be transformed into a library that could be used by other processes. This would require minor transformations of the initialiser to not read the state of the system from the boot information passed by the kernel, but there are no conceptually hard parts in doing this.

Take-grant protection model

Our formalisation of the take-grant protection model does not distinguish between active *subjects* and passive *objects*, but treats both as active *entities*, which can lead to an overestimation of authority confinement and information flow. The model could be refined by making a distinction between the two.

8.4 Concluding remarks

The security of an operating system depends on setting it up correctly. The authority confinement, integrity, and information flow proofs for seL4 (Sewell et al., 2011; Murray et al., 2013) show that if the state of a system is initialised in conformance with an access control policy, then seL4 will enforce this policy.

Often code that does this initialisation is ad-hoc and manually written, with little guarantee about correctness. Instead, in this thesis we have presented a formal algorithm of a generic system initialiser that starts a system in conformance with a given specification.

The specifications of systems are described using the capability distribution language, capDL, which Boyton et al. (2013) formally linked to the access control policies of the authority confinement, integrity, and information flow proofs for seL4, allowing us to reason about the security of a system by examining the capDL specification of a system.

We have shown the about the correctness of this formal algorithm, proving that each step of our algorithm correctly creates and configures the objects described, while not undoing the work done by the previous steps. To do this, we have developed a custom separation logic that is used to reason both about the correctness of our formal system initialisation algorithm and to develop a formal API specification for the seL4 kernel itself.

By connecting the proof of correctness of our system initialisation algorithm to the existing proofs of the formally verified seL4 microkernel, and using the capDL language to specify our system, we can have a very strong confidence in the correctness of our algorithm, knowing that we can start a system correctly and that seL4 will enforce the security of our system as specified.

Bibliography

Alves-Foss, Jim et al. (2006). 'The MILS Architecture for High-Assurance Embedded Systems'. In: *Int. J. Emb. Syst.* 2, pp. 239–247.

Andronick, June, David Greenaway, and Kevin Elphinstone (2010). 'Towards proving security in the presence of large untrusted components'. In: *5th SSV*.

Andronick, June, Andrew Boyton, and Gerwin Klein (2012). *Final Report for AOARD Grant #FA2386-11-1-4070, Formal System Verification - Extension*. Technical Report. NICTA.

Barth, Adam et al. (2008). *The security architecture of the Chromium browser*. URL: http://crypto.stanford.edu/websec/chromium/chromium-security-architecture.pdf.

Bishop, Matt (1996). 'Conspiracy and Information Flow in the Take-Grant Protection Model'. In: *Journal of Computer Security* 4.4, pp. 331–360.

Bishop, Matt and Lawrence Snyder (1979). 'The transfer of information and authority in a protection system'. In: *7th SOSP*, pp. 45–54. ISBN: 0-89791-009-5.

Bornat, Richard (2000). 'Proving pointer programs in Hoare Logic'. In: *5th MPC*. Vol. 1837. LNCS, pp. 102–126. DOI: 10.1007/10722010_8.

Bornat, Richard et al. (2005). 'Permission Accounting in Separation Logic'. In: *SIGPLAN Not.* 40.1, pp. 259–270. DOI: 10.1145/1047659.1040327.

Boyton, Andrew (2009). 'A Verified Shared Capability Model'. In: *4th International Workshop on Systems Software Verification*, pp. 25–44.

Boyton, Andrew et al. (2013). 'Formally Verified System Initialisation'. In: *15th ICFEM*, pp. 70–85. DOI: 10.1007/978-3-642-41202-8_6.

Boyton, Andrew et al. (2014). *capDL initialiser+loader, version 1.0.0*. DOI: 10.5281/zenodo.11254.

Burstall, Rod (1972). 'Some techniques for proving correctness of programs which alter data structures'. In: *Machine Intelligence 7*, pp. 23–50.

Calcagno, Cristiano, Peter W. O'Hearn, and Hongseok Yang (2007). 'Local Action and Abstract Separation Logic'. In: *Proc. 22nd LICS*, pp. 366–378. DOI: 10.1109/LICS.2007.30.

Cock, David, Gerwin Klein, and Thomas Sewell (2008). 'Secure Microkernels, State Monads and Scalable Refinement'. In: *21st TPHOLs*, pp. 167–182. DOI: 10.1007/978-3-540-71067-7_16.

Dang, H.-H., P. Höfner, and B. Möller (2011). 'Algebraic separation logic'. In: *The Journal of Logic and Algebraic Programming* 80.6. Relations and Kleene Algebras in Computer Science, pp. 221 –247. DOI: 10.1016/j.jlap.2011.04.003.

Dennis, Jack B. and Earl C. Van Horn (1966). 'Programming Semantics for Multiprogrammed Computations'. In: *CACM* 9, pp. 143–155.

Dockins, Robert, Aquinas Hobor, and Andrew W. Appel (2009). 'A Fresh Look at Separation Algebras and Share Accounting'. In: *Programming Languages and Systems*. Vol. 5904. Lecture Notes in Computer Science, pp. 161–177. DOI: 10.1007/978-3-642-10672-9_13.

Elkaduwe, Dhammika, Philip Derrin, and Kevin Elphinstone (2006). 'Kernel Data – First Class Citizens of the System'. In: *2nd Int. WS Obj. Syst. & Softw. Arch.* Pp. 39–43.

— (2007). 'A Memory Allocation Model for an Embedded Microkernel'. In: *1st MIKES*. NICTA, pp. 28–34.

Elkaduwe, Dhammika, Gerwin Klein, and Kevin Elphinstone (2008). 'Verified Protection Model of the seL4 Microkernel'. In: *VSTTE 2008*. Vol. 5295. LNCS, pp. 99–114.

Filliâtre, Jean-Christophe and Claude Marché (2007). 'The Why/Krakatoa/Caduceus Platform for Deductive Program Verification'. In: *Computer Aided Verification*. Vol. 4590. Lecture Notes in Computer Science, pp. 173–177. DOI: 10.1007/978-3-540-73368-3_21.

Gordon, Michael J. C., Robin Milner, and Christopher P. Wadsworth (1979). *Edinburgh LCF*. Vol. 78. LNCS. Springer. ISBN: 3-540-09724-4.

Gordon, Mike (2000). 'Proof, Language, and Interaction'. In: chap. From LCF to HOL: A Short History, pp. 169–185. ISBN: 0-262-16188-5.

Greenaway, David, June Andronick, and Gerwin Klein (2012). 'Bridging the Gap: Automatic Verified Abstraction of C'. In: *3rd ITP*. Vol. 7406. LNCS, pp. 99–115. ISBN: 978-3-642-32346-1.

Greenaway, David et al. (2014). 'Don't Sweat the Small Stuff: Formal Verification of C Code Without the Pain'. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 429–439. DOI: 10.1145/2594291.2594296.

Greenhills Software, Inc. (2008). *Integrity Real-Time Operating System*. Accessed August 2014. URL: http://www.ghs.com/products/rtos/integrity.html.

Harrison, Michael A., Walter L. Ruzzo, and Jeffrey D. Ullman (1976). 'Protection in Operating Systems'. In: *CACM*, pp. 561–471.

Hicks, Boniface et al. (2007). 'A logical specification and analysis for SELinux MLS policy'. In: *SACMAT*, pp. 91–100. DOI: 10.1145/1266840.1266854.

Hoare, C. A. R. (1969). 'An axiomatic basis for computer programming'. In: *CACM* 12.10, pp. 576–580.

iOS 7: Understanding Location Services (2014). Accessed August 2014. Apple. URL: http://support.apple.com/kb/HT5594.

iOS Security (2014). White Paper. Apple.

Jensen, Jonas Braband and Lars Birkedal (2012). 'Fictional Separation Logic'. In: *Programming Languages and Systems*. Vol. 7211. Lecture Notes in Computer Science, pp. 377–396. DOI: 10.1007/978-3-642-28869-2_19.

Jones, A. K., R. J. Lipton, and L. Snyder (1976). 'A Linear time algorithm for deciding security'. In: *Foundations of Computer Science, 1976., 17th Annual Symposium on*, pp. 33–41. DOI: 10.1109/SFCS.1976.1.

Kauer, Bernhard (2007). 'OSLO: Improving the Security of Trusted Computing'. In: *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*. SS'07, 16:1–16:9. ISBN: 111-333-5555-77-9.

Klein, Gerwin et al. (2009). 'seL4: Formal Verification of an OS Kernel'. In: *SOSP*, pp. 207–220. DOI: 10.1145/1629575.1629596.

Klein, Gerwin, Rafal Kolanski, and Andrew Boyton (2012). 'Mechanised Separation Algebra'. In: *Interactive Theorem Proving (ITP)*, pp. 332–337.

Klein, Gerwin et al. (2014). 'Comprehensive Formal Verification of an OS Microkernel'. In: *Trans. Comp. Syst.* 32.1, 2:1–2:70. DOI: 10.1145/2560537.

Kolanski, Rafal (2011). 'Verification of Programs in Virtual Memory Using Separation Logic'. Available from publications page at http://ssrg.nicta.com.au/. PhD thesis. UNSW.

Kuz, Ihor et al. (2007). 'CAmkES: A component model for secure microkernel-based embedded systems'. In: *Journal of Systems and Software Special Edition on Component-Based Software Engineering of Trustworthy Embedded Systems* 80.5, pp. 687–699. DOI: 10.1016/j.jss.2006.08.039.

Kuz, Ihor et al. (2010). 'capDL: A Language for Describing Capability-Based Systems'. In: *1st APSys*, pp. 31–36.

Lipton, Richard J. and Lawrence Snyder (1977). 'A Linear Time Algorithm for Deciding Subject Security'. In: *J. ACM* 24.3, pp. 455–464. DOI: http://doi.acm.org/10.1145/322017.322025.

Martin, W. B., P. D. White, and F. S. Taylor (2002). 'Creating High Confidence in a Separation Kernel'. In: *Automated Softw. Engin.* 9.3, pp. 263–284. URL: http://portal.acm.org/citation.cfm?id=592088.

Murray, Toby et al. (2013). 'seL4: from General Purpose to a Proof of Information Flow Enforcement'. In: *IEEE Symp. Security & Privacy*, pp. 415–429. DOI: 10.1109/SP.2013.35.

National Security Agency (2007). *U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness, Version 1.3*.

Nipkow, Tobias, Lawrence Paulson, and Markus Wenzel (2002). *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Vol. 2283. LNCS. Springer. DOI: 10.1007/3-540-45949-9.

Norrish, Michael (1998). 'C formalised in HOL'. PhD thesis. University of Cambridge Computer Laboratory.

— (2013). *C-to-Isabelle Parser, version 1.13.0*. Accessed August 2014. URL: http://ertos.nicta.com.au/software/c-parser/.

Norvig, Peter (2006). *Extra, Extra — Read All About It: Nearly All Binary Searches and Mergesorts are Broken, Google Research Blog*. Accessed August 2014. URL: http://googleresearch.blogspot.com.au/2006/06/extra-extra-read-all-about-it-nearly.html.

Open Kernel Labs (2008a). *Elfweaver Reference Manual*. URL: http://wiki.ok-labs.com/downloads/release-2.1.1-patch.9/elfweaver-user-manual_2.1.1.pdf.

— (2008b). *OKL4 Microkernel, Reference Manual*. URL: http://wiki.ok-labs.com/downloads/release-3.0/okl4-ref-manual-3.0.pdf.

Parkinson, Matthew J. (2005). 'Local Reasoning for Java'. PhD thesis. Computer Laboratory. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-654.html.

Parkinson, Matthew J. and Gavin M. Bierman (2008). 'Separation Logic, Abstraction and Inheritance'. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08, pp. 75–86. DOI: 10.1145/1328438.1328451.

Reynolds, John C. (2002). 'Separation Logic: A Logic for Shared Mutable Data Structures'. In: *Proc. 17th IEEE Symposium on Logic in Computer Science*, pp. 55–74.

Rushby, John M. (1981). 'Design and Verification of Secure Systems'. In: *8th SOSP*, pp. 12–21. DOI: 10.1145/800216.806586.

Rushby, John (2009). 'Software verification and system assurance'. In: *Software Engineering and Formal Methods, 2009 Seventh IEEE International Conference on*. IEEE, pp. 3–10.

Saaltink, Mark (1997). 'Domain checking Z specifications'. In: *4th NASA Formal Methods Workshop*, pp. 185–191.

Saltzer, Jerome H. (1974). 'Protection and the Control of Information Sharing in Multics'. In: *CACM* 17, pp. 388–402.

Sewell, Thomas et al. (2011). 'seL4 Enforces Integrity'. In: *2nd ITP*. Vol. 6898. LNCS, pp. 325–340. DOI: 10.1007/978-3-642-22863-6_24.

Sewell, Thomas, Magnus Myreen, and Gerwin Klein (2013). 'Translation Validation for a Verified OS Kernel'. In: *PLDI*, pp. 471–481.

Shapiro, Jonathan S. (1999). 'EROS: A Capability System'. PhD thesis. University of Pennsylvania. URL: http://www.eros-os.org/papers/shap-thesis.ps.

Shapiro, Jonathan S. and Norman Hardy (2002). 'EROS: A Principle-Driven Operating System from the Ground Up'. In: *IEEE Software* 19.1, pp. 26–33. DOI: 10.1109/52.976938.

Shapiro, Jonathan S. and Sam Weber (2000). 'Verifying the EROS Confinement Mechanism'. In: *IEEE Symposium on Security and Privacy*, pp. 166–176. DOI: 10.1109/SECPRI.2000.848454.

Singaravelu, Lenin et al. (2006). 'Reducing TCB Complexity for Security-Sensitive Applications: Three Case Studies'. In: *1st EuroSys*, pp. 161–174. DOI: 10.1145/1217935.1217951.

Snyder, Lawrence (1977). 'On the synthesis and analysis of protection systems'. In: *ACM SIGOPS Operating Systems Review* 11.5, pp. 141–150.

— (1981a). 'Formal models of capability-based protection systems'. In: *Computers, IEEE Transactions on* 100.3, pp. 172–181.

— (1981b). 'Theft and Conspiracy in the Take-Grant Protection Model'. In: *Journal of Computer and System Sciences* 23.3, pp. 333–347.

Trustworthy Systems Team (2014). *seL4 proofs for API 1.03, release 2014-08-10*. DOI: 10.5281/zenodo.11248.

Tuch, Harvey (2008). 'Formal Memory Models for Verifying C Systems Code'. PhD thesis. UNSW.

Verbeek, Freek et al. (2014). 'Formal Specification of a Generic Separation Kernel'. In: *Archive of Formal Proofs*. Formal proof development. ISSN: 2150-914X. URL: http://afp.sf.net/entries/CISC-Kernel.shtml.

Wu, Chunhan, Xingyuan Zhang, and Christian Urban (2013). 'A Formal Model and Correctness Proof for an Access Control Policy Framework'. In: *Certified Programs and Proofs*. Vol. 8307. Lecture Notes in Computer Science, pp. 292–307. DOI: 10.1007/978-3-319-03545-1_19.

Yang, Hongseok and Peter W. O'Hearn (2002). 'A Semantic Basis for Local Reasoning'. In: *Foundations of Software Science and Computation Structure*, pp. 402–416.