School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Refining seL4's Accounting of Touched Addresses for Time Protection

by

Thomas Liang

Thesis submitted as a requirement for the degree of

Bachelor of Advanced Computer Science

Submitted: November 2025

Supervisor: Dr. Rob Sison

Student ID: z5418574

Abstract

Given that a Haskell program and a C program have the same behaviour, and the knowledge that the Haskell program only accesses addresses in a certain partition, how can you prove that the C program does also? *Time Protection* is a recently proposed and implemented set of mechanisms for the seL4 microkernel. It eliminates *microarchitectural timing channels*, the leaking of information through the timing of hardware events and also a major threat to computer security. To verify Time Protection, we must take the existing refinement proof—from seL4's intermediate Haskell implementation to its true C implementation—and extend it to show that its *touched addresses set* does not stray from a defined partition. I give a summary of the verification of seL4 and Time Protection so far, give a formalisation of touched addresses, and then demonstrate a proof method on a small part of the kernel. Finally, I give an argument that the methods can be scaled to verify the rest of the kernel.

Acknowledgements

I thank my supervisor Rob Sison for their endless patience and reassurance throughout this project, and reviewing countless drafts. I always felt supported. I also thank my assessor Thomas Sewell for sharing his veritable treasure trove of technical advice about Isabelle/HOL and the proof frameworks I used.

Gerwin Klein helped me with understanding the structure of the proof-base, and advised the assertion-based approach I propose. Julia Vassiliki guided the choice of which part of the kernel to demonstrate my methods on.

Lastly, I thank my partner Julia for her unwavering encouragement throughout this project, and keeping me company as I wrote this thesis.

List of Figures

2.1 A proof tree for Example 2.1. 13

4.1 The new call-tree for partition-guard variants 31
4.2 A specification lemma for `getSlotCap` 33
4.3 Operation dependency tree for `lookupCap` 38
4.4 The Haskell definitions of `getSlotCapInDomPart` and `getCTEInDomPart` . . 39
4.5 The definition of `updateCapInDomPart`. 44
4.6 The definition of `cap_untyped_cap_set_capFreeIndex`. 44

List of Theorems

2.1 Example (Demonstration of composing preconditions) 11

Correspondence step 12

Preconditions step 12

2.2 Proposition (Partition subset invariant [Buckley et al., 2023, p.11]) 16

2.3 Definition (Memory usage WRT a certain state change [Verbeek et al., 2020, p. 103]) 18

2.4 Definition (Memory usage Hoare triple [Verbeek et al., 2020, p. 104]) 18

3.1 Proposition (The Partition Oracle) 23

3.2 Proposition (Structural similarity of kernel object accesses) 24

4.1 Definition (The partition oracle) 28

4.2 Lemma (Domain partitions do not overlap) 28

4.3 Definition (Concrete specification–partition guard) 28

4.4 Definition (Executable specification–partition guard) 30

4.5 Definition (Partition guard–variants for fetching and updating objects) . . . 30

4.6 Lemma (Predicate for moving concrete specification–guards) 33

4.7 Lemma (Move concrete specification–guards into preconditions) 33

4.8 Lemma (Move concrete specification–partition guards) 34

4.9 Theorem (Partition guard–introduction) 34

Correspondence step 36

Preconditions step 36

4.10 Theorem (Correspondence statement for getSlotCap) 38

4.11 Lemma (Rewrite getSlotCapInDomPart in terms of stateAssert) 39

4.12 Lemma (Upgrading lemma for getSlotCap) 40

Correspondence step 40

Preconditions step 40

4.13 Theorem (Correspondence of getSlotCapInDomPart) 41

4.14 Proposition (Correspondence of resolveAddressBitsInDomPart) 42

4.15 Proposition (Specification lemma of resolveAddressBitsInDomPart) 42

4.16 Lemma (Calling getSlotCapInDomPart does not mutate the state) 42

4.17 Lemma (Specification lemma for `getSlotCapInDomPart`) 42

4.18 Lemma (Specification lemma for `lookupSlotForThreadInDomPart`) 43

4.19 Theorem (Correspondence of lookupSlotForThreadInDomPart) 43

4.20 Theorem (Correspondence of lookupCap) 43

4.21 Lemma (Rewrite updateCapInDomPart in terms of stateAssert) 45

4.22 Lemma (Upgrading lemma for updateCapInDomPart) 45

4.23 Theorem (Correspondence statement for updateCapInDomPart) 45

B.1 Lemma (If the left-hand side would fail, assume it does not) 57

B.2 Lemma (Generalisation of correspondence preconditions) 57

B.3 Lemma (A left-hand side assertion corresponds to doing nothing) 57

B.4 Lemma (Remove a Skip at the end of the right-hand side) 57

B.5 Lemma (Remove a return at the end of the left-hand side) 57

Contents

1 Connection to the Time Protection project **1**

1.1 Overview of contributions . . . 3

2 Review of related formal methods **4**

2.1 Assumed knowledge . . . 4

2.2 Correctness of seL4 is verified using data refinement . . . 5

2.3 Confidentiality in seL4 . . . 6

2.4 Key concepts of the seL4 microkernel . . . 6

2.5 Representation of the Haskell implementation . . . 7

2.6 Embedding the C implementation . . . 7

2.7 Refinement from the executable to concrete specification . . . 8

2.7.1 State relations . . . 9

2.7.2 Correspondence statements . . . 9

2.7.3 Failure in the executable specification . . . 10

2.7.4 Composing correspondence statements . . . 10

2.8 Timing channels in data cache . . . 14

2.9 How does time protection work? . . . 14

2.10 Verifying time protection enforcement . . . 15

2.11 Related work . . . 16

2.11.1 Tracking the touched addresses set for the abstract specification . . . 17
2.11.2 Verifying isolation of an operating system with refinement 17
2.11.3 Capturing touched addresses for assembly verification 18
2.12 Summary 19

3 Formalised accounting of touched addresses 20
3.1 Restatement of the goals of this project 20
3.2 Adding reasoning about touched addresses 21
3.3 Inserting the partition guards into the concrete specification 22
3.4 Refining from the executable specification 23
3.5 Choosing a subset of the kernel 25
3.6 Summary 26

4 Refinement of touched addresses on a simple part of the kernel 27
4.1 Definitions and assumptions 27
4.1.1 The partition oracle 28
4.1.2 Partition guards in the concrete specification 28
4.1.3 Partition guards in the executable specification 30
4.2 An introduction rule for partition guards 33
4.3 Repairing correspondence of a simple procedure 37
4.3.1 Extending correspondence to guarded variants 38
4.3.2 An additional assumption 42
4.3.3 Extending specification lemmas for guarded variants 42
4.3.4 Finishing the top-level proof 43
4.4 Testing the methods on another procedure 44
4.5 Summary 46

5 Discussion of scalability of the proposed proof methods **47**

5.1 How much effort did these proofs take? 47

5.2 Do these techniques apply to other parts of the kernel? 48

5.3 Summary 49

6 Conclusion **50**

6.1 What comes next? 50

6.2 Closing remarks 52

Bibliography **53**

Notation used in this thesis **56**

Useful existing proved lemmas **57**

Chapter 1

Connection to the Time Protection project

This project is part of a young program to produce the first ever proof that an operating system (OS) kernel is free of a subtle type of security vulnerability called a *timing channel*. A kind of *covert channel*—a mechanism in a system that is not intended to be used for communication [Lampson, 1973]—timing channels leak information about the actions of your machine that is not read from a secret buffer, but through the time it takes to execute certain operations. Specifically, we are concerned with *microarchitectural timing channels* (µTCs), which are timing channels that carry information as a result of processor state that is hidden from programs by design, and is absent from the instruction set architecture [Ge et al., 2018]. This state includes the state of the branch predictor, prefetcher, TLBs, and most critical for this project, data caches.

The nature of timing channels means that existing OS mechanisms that prevent the flow of information, such as partitioning physical memory into separate virtual address spaces, are insufficient to block them. Two processes, sharing no memory, files, pipes, signals, or any other OS communication mechanism, can still communicate (albeit slowly) by sending messages over a timing channel in your hardware. (I shall outline one such attack in Section 2.8.) Timing channels pose a huge risk, as we live in a world where we run code without inspecting it on our machines all the time. Software developers will download and run library dependencies without much consideration. Your web browser runs thousands of lines of JavaScript code you will never read, for every page you visit. Critical services run on the cloud in virtual machines, alongside potentially thousands of other untrustworthy virtual machines on the same hardware.

In fact, you may have already heard about µTCs and been affected by them without being aware of it. The disclosure of the Meltdown [Lipp et al., 2018] and Spectre [Kocher et al., 2019] attacks shocked the world, as nearly every modern processor from major manufacturers was vulnerable [Metz and Perlroth, 2018]. These attacks put arbitrary kernel memory in cache, such that it is readable through a cache timing attack. If we want to be assured that our computer systems are secure, it is crucial that we eliminate µTCs.

The most critical line of defence against security vulnerabilities is the kernel, so how might we prevent µTCs there? Ge et al. [2019] implemented the set of OS mechanisms called *time protection* to fundamentally eliminate µTCs for the seL4 microkernel [Klein et al., 2009]. The mechanisms carefully use available hardware mechanisms on x86 and Arm platforms to partition and flush microarchitectural state. The authors show empirically that the actions of one component of the operating system can hidden to other components for the most part, with minimal performance overhead. Remaining microarchitectural state not dealt with are addressed by Wistoff et al. [2021, 2023], who introduce a new instruction for an open source Risc-V processor core. Then, Buckley et al. [2023] ports the Time Protection mechanisms to take advantage of the new instruction, to eliminate all known µTCs. Together, these works show that elimination of µTCs for real-world kernels is feasible, and reinforces seL4 position as the world's most secure microkernel.

To have total assurance that the time protection mechanisms in seL4 are correct, however, we must *formally verify* that seL4 eliminates µTCs. Formal verification is the process of applying mathematical techniques to produce a *computer-checked proof* that certain properties hold about our system. Computer-checked proofs exist that show that seL4 is free of errors and always behaves correctly (so-called "functional correctness"), while enforcing security properties like integrity, availability, and confidentiality [Heiser, 2020]. As mentioned at the beginning, there is a project to extend these security proofs to also address the threat of timing channels, which this project is a part of. If completed, it would be a result that is the first of its kind. This is referred to in this thesis as "verifying Time Protection".

Formal verification of time protection has had strong progress so far, but it hinges on the answer to a currently open research question: Does the kernel itself obey the security policy? I build on work by Buckley et al. [2023]; Sison et al. [2023], who formalise the notion of preventing µTCs by an OS, and apply it in the context of seL4. The authors theorise that verifying Time Protection is possible by giving an *extension* to the existing seL4 proof base that produces a proof of time protection enforcement. To use this extension, we must give it a proof that the kernel only dereferences memory addresses in a particular *partition*, determined by the OS security policy, at certain times.[1] A proof of this invariant does not yet exist, but is the goal of this project and many others.

This thesis presents one part of the story of proving this key invariant: what addresses the kernel will access when it is called. I will develop a method for reasoning about the above invariant at the level of the C implementation, given assumptions about a higher-level model of the kernel implemented in Haskell. In doing so, I reduce the burden of verifying the C implementation, by *reducing* it to a problem at the more abstract, and easier to reason-about Haskell implementation. Recent developments between the seL4 proof maintainers and kernel developers suggest a promising new direction for verifying time protection, (that could even link with future proofs about the multikernel version of seL4). As a result, we do away with the approach proposed by Buckley et al. [2023] to track what addresses the kernel accesses. However, their foundational ideas still apply.

In conjunction with a series of other projects (particularly Nair [2025]), we endeavour to

[1] Called the "partition subset invariant" in Buckley et al. [2023], and is discussed later as Proposition 2.2.

demonstrate a comprehensive method that, if followed to completion, can be used with the framework of Buckley et al. [2023] to prove that seL4 correctly enforces time protection.

1.1 Overview of contributions

I make the following contributions:

- a survey of progress on verifying time protection, and more importantly the new proposed direction for doing so (Chapter 2);
- a proposed method for reasoning about what addresses the kernel will access when called, during refinement from the intermediate Haskell implementation to the concrete specification (Chapter 3);
- a demonstration of these methods on a small subset of the kernel (Chapter 4);
- an argument that these methods are scalable to verification of the entire kernel (Chapter 5); and lastly
- close with some hypotheses for how that might be done (Chapter 6)

Chapter 2

Review of related formal methods

This chapter outlines the prerequisite knowledge that underlies the rest of the thesis. I shall

- Give an overview of seL4's formal verification, and the techniques that are applied for those proofs (Sections 2.2, 2.5 and 2.6),
- Discuss the proof of confidentiality for seL4, a key result that Time Protection is extending (Section 2.3),
- Clarify some key concepts about the design of seL4 (Section 2.4),
- Introduce the "backwards-reasoning" style of proof used to connect seL4's executable and concrete specifications (Section 2.7),
- Give a glimpse of the problem that Time Protection seeks to solve (Sections 2.8 to 2.10), and lastly
- Compare this thesis with previous work, including that of Buckley et al. [2023] (Section 2.11).

The informed reader should feel free to skip sections as they please. However, Section 2.7 is particularly important to understanding the rest of the proofs in this thesis, as I sketch proofs in an esoteric way.

2.1 Assumed knowledge

I will assume that the reader has an understanding of operating system fundamentals (see COMP3231 Operating Systems), basic knowledge about/ software verification (see COMP4161 Advanced Topics in Software Verification), and some familiarity with the seL4 microkernel (see Heiser [2020]; Trustworthy Systems Team [2017]). I also assume that the reader is generally familiar with the C and Haskell programming languages, but any listings will be accompanied

by a brief explanation. I will briefly explain the most relevant concepts in this chapter, and leave references to further reading, such as the seL4 documentation [Trustworthy Systems Team, 2017].

2.2 Correctness of seL4 is verified using data refinement

Before we look at the existing work on verifying time protection, let us review how the verification of seL4 has proceeded so far and where this project will fit into it. All the following work is conducted in the Isabelle/HOL interactive theorem prover [Wenzel, 2014].

The initial functional correctness proof of seL4 [Klein et al., 2009] uses a proof technique called **data refinement** [de Roever and Engelhardt, 1998], abbreviated to just **refinement**. We say that a low-level description of a system's behaviour *refines* a high-level description if every possible behaviour of the low-level specification is also a possible behaviour of the high-level description. Rather than low-level or high-level, we say it is more *concrete* or more *abstract* respectively. We will continue to use the spatial metaphor however, as we say refinement carries *down* properties.[1] Refinement is a transitive property: if A refines B, and B refines C, then A also refines C. The beauty of refinement is that any property that we can prove about the abstract description can be carried down to apply to the concrete description as well. Hence, when we are reasoning over a complicated system written in a language with complicated semantics–i.e. the C implementation of seL4—we can instead reason over an abstract model of the system in simpler mathematical terms.

There are three descriptions of seL4 in the refinement chain. At the top of the refinement chain is the **abstract specification**. The abstract specification is hand-written and is declarative, saying only what the kernel does but not anything about the data structures or algorithms that would implement the behaviour. The classic example is scheduling. The scheduler is only specified to pick *a* runnable thread, as opposed to the round-robin scheduler in the C implementation. Below the abstract specification, there is the **executable specification**, derived by a mechanical process from a Haskell implementation of the kernel. Its purpose is to elaborate on how kernel objects and operations are implemented, while still avoiding the details of the low-level optimisations present in the C implementation. The scheduler of the executable specification will select the first runnable thread in the list of threads, that has the highest priority. At the bottom there is the **concrete specification**, derived by a machine process from the C implementation and is the most detailed of the three specifications. I describe the executable specification and concrete specification in more detail in Section 2.5 and Section 2.6 respectively, as this project focuses on the refinement between them.

Past work proved that the executable specification refines the abstract, and that the concrete specification refines the executable [Klein et al., 2009]. By transitivity, we have the concrete specification refines the abstract. Thus, we have a machine-checked guarantee that the C

[1] Behaviours include when a system would raise an error. In any scenario, if A refines B and B would not fail, then neither does A.

implementation of the kernel behaves only in the ways described by the specification.[2] This is called the **functional correctness** proof of seL4.

2.3 Confidentiality in seL4

However, adherence to a specification is useless if the specification itself is flawed, so there are also proofs about the abstract specification that demonstrate that seL4 enforces crucial security properties [Heiser, 2020]. Of particular interest to us is the information-flow proofs that the kernel enforces *confidentiality* [Murray et al., 2013]. These security properties are then carried down to apply to the C implementation via refinement.

Confidentiality says that a "domain" cannot learn any information about—i.e. observe the effects of—the actions of other domains, unless an explicit communication channel has been established between them.[3] Roughly speaking, a **domain** is a collection of threads that should be "isolated" in a security sense. There is always one currently-running domain, and only threads belonging to the current domain can be scheduled. What domains exist and the order they are scheduled is fixed and compiled with the kernel [Trustworthy Systems Team, 2017].

I emphasise that these proofs do not eliminate information flow *in general*. Rather, they require the system designer to first isolate untrustworthy components of the OS into their own domains. Then, seL4 ensures that no channels exist between domains.

Note also that the current proof of confidentiality is limited to **storage channels**, where information is carried through writing to storage locations. The purpose of verifying time protection to extend confidentiality to cover timing channels in addition to storage channels. Traditionally, all covert channels are classified as either storage or timing channels [DoD 1986]. Hence, verifying Time Protection would eliminate all covert channels.

2.4 Key concepts of the seL4 microkernel

Let us briefly define some key terminology we use about seL4. I will use these terms to describe the internals of the kernel, and in particular, the case studies in Chapter 4. The seL4 microkernel is a capability-based OS kernel.

A capability is a kernel object that acts like a key: it entitles the holder some form of access (read, or write, etc.) to another kernel object, such as a Notification or an Endpoint. Users do not interact with capabilities directly, but rather through **capability pointers** (CPtr). Capabilities are stored in **capability table entries** (CTE) slots, which are arranged in tree structures

[2]The functional correctness proof of seL4 extends to the compiled binary as well, as developed by Sewell [2017]. The binary could be considered a specification, but only the above three specifications are the most important for our purposes.

[3]An example of an explicit communication channel would be through a Notification object.

known as **capability spaces** (CSpace). For a proper treatment, please see the seL4 Manual [Trustworthy Systems Team, 2017].

2.5 Representation of the Haskell implementation

Before we can do any work on any implementation of a program, we must first translate it into data types inside the theorem prover. This section will briefly describe how the intermediate Haskell implementation of seL4 is translated into and *deeply embedded* in the *executable specification.*

The Haskell implementation of seL4 is translated into Isabelle/HOL as monadic computations on a monad developed by Cock et al. [2008]. [4] The monad $'a$ kernel is a state monad—to model computations which may modify the kernel state—equipped with a failure flag to model Haskell code raising an error, and returns a set of states to model nondeterminism. For a function that returns $'a$ kernel, you can read the $'a$ of as the type of the return value of the function. For example, the type signature $'a$ list → machine-word kernel is a function that given a list of words returns a word, depending on the state of the kernel and potentially modifying it. The monad is isomorphic to k-state ⇒ ($'a$ × k-state) set × bool.

The kernel state k-state is a record of global variables. These include: ksPSpace, the state of the heap; ksCurThread, a pointer to the TCB object for the currently-running thread; and most important for our purposes, ksCurDomain, an 8-bit integer which identifies the currently-running domain. Because the executable specification is a deep embedding, local variables in the Haskell code are represented using higher order syntax, as variable bindings in Isabelle/HOL.

Cock et al. [2008] also defines a Hoare logic for proving properties about the Haskell specification. The details of the logic are not crucial here, but we shall occasionally see statements phrased using the Hoare logic.

For a more detailed treatment, see Cock et al. [2008].

2.6 Embedding the C implementation

The C implementation is modelled using a language called SIMPL, developed by Schirmer [2006]. It is a *shallow embedding* and has the following syntax (reproduced from Winwood et al. [2009]):

$$\begin{aligned} c \triangleq\ & \mathsf{Skip} \mid {\acute{}}v\ :== e \mid c_1\ ;;\ c_2 \mid \mathsf{IF}\ e\ \mathsf{THEN}\ c_1\ \mathsf{ELSE}\ c_2\ \mathsf{FI} \mid \mathsf{WHILE}\ e\ \mathsf{DO}\ c\ \mathsf{OD} \\ & \mid \mathsf{TRY}\ c_1\ \mathsf{CATCH}\ c_2\ \mathsf{END} \mid \mathsf{THROW} \mid \mathsf{Call}\ f \mid \mathsf{Guard}\ F\ P\ c \end{aligned}$$

[4]However, we present it here in a way derived from Winwood et al. [2009].

The semantics of each piece of syntax is what you would expect for an imperative programming language. To embed C, Winwood et al. [2009] instantiate Simpl with a model of C semantics and a C memory model to form C-Simpl. The C parser embeds the C code into C-Simpl to form the concrete specification. We reuse the verification condition generator by Schirmer [2006] to verify instances of C code.

Note that function calls are represented as Call f where f is some symbol that can be mapped to a function definition. This is opposed to the executable specification, where functions are Isabelle/Hol functions, and so calling a function is function application in Isabelle/Hol.

For our purposes, the most important kind of syntax are the Guard statements. To evaluate a statement Guard $F\,P\,c$, we check whether the proposition P holds. If it is false, we set the failure flag of the nondeterministic state monad, which stops the program. Only if it is true, do we continue to execute the Simpl statement c. The component F denotes the kind of fault the guard is checking.

You might notice that there is no "return" statement. All control flow in Simpl is modelled with exceptions; function bodies are surrounded by a TRY block, and to return early from a function, it "throws" the correct exception.

Variables are either local or global variables. Global variables are handled the same as they are in the executable specification, as fields in a state record. The fields have slightly different types than they do in the executable specification; namely that it is more efficient to store data as full machine words in C, so fields such as the currently-running domain (ksCurDomain) is stored as a 64-bit integer. To get the global state record, we project the state with globals. To give an example,

$$\text{ksCurDomain (globals } s)$$

would be the identifier of the currently-running domain in state s. Local variables are fields of a state record, automatically generated by the C parser.

For more details, see Winwood et al. [2009].

Lastly, we ought to review the notation used for the heap. The heap is denoted with $\mathscr{H}$, such that accessing a pointer p to the heap is denoted $\mathscr{H}\ p$, and updating the heap at pointer p to be value v is denoted $\mathscr{H}(p \mapsto v)$. To access a field f of a struct we write $\&(p \to f)$. For a comprehensive description of the model of the heap, see [Tuch et al., 2007].

2.7 Refinement from the executable to concrete specification

How do we actually show that a C program is a refinement of a Haskell program? There is a rich literature about (data) refinement, the canonical text being de Roever and Engelhardt [1998] from whom we have borrowed terminology.[5] The seL4 refinement proofs use a proof method

[5] As a historical aside, data refinement is first published by Milner [1970, 1971], which are themselves synthesised from a series of internal memos written by Milner.

that is a variation of *forward simulation* [Winwood et al., 2009], a synonym for *L-simulation* [de Roever and Engelhardt, 1998]. In forward simulation, we must show that a step in the concrete specification leads to states that a step in the abstract specification could lead to. In this section, we will briefly cover the tools and techniques used for refinement from Haskell to C for seL4. For a detailed treatment of the correspondence framework, see Winwood et al. [2009].

2.7.1 State relations

The first step of a refinement proof is to define a *state relation*[6] between the states of your abstract program—the executable specification—and the concrete program—the concrete specification. For refinement from Haskell to C, that relation is called sr-rf. Its definition is too long to present here, but in short, if we have that $(s, s') \in \text{sr-rf}$ then the global kernel state of s and s' are semantically equivalent. For example, for the currently-running domain, we have that

$$(s, s') \in \text{sr-rf} \implies \text{ksCurDomain } s = \text{ucast } (\text{ksCurDomain } s')$$

where the right-hand side contains an integer cast because the current domain is a 8-bit integer on the left, and a 64-bit integer on the right.

2.7.2 Correspondence statements

Refinement is often expressed as a **correspondence statement** between a Haskell and C operation. A correspondence statement looks like

$$\text{ccorres } r\ xf\ \ P\ P'\ \ hs\ a\ c.$$

and each argument can be read as follows.

- The return values of a and c are related by **return value relation** r
- Because return values in C are modelled as updating a local return variable, we need to describe how to get the return value using the **extraction function** xf
- P and P' are the **preconditions** that must hold, about the Haskell and C states respectively, for the correspondence to be true
- The hs parameter is a **handler stack**. Control flow in C is modelled as throwing and catching exceptions (See Section 2.6). This parameter allows us to provide a list of exception handlers for the C operation
- Lastly, a and c are the Haskell and C operations. These are referred to as the **left-hand side** and **right-hand** sides.

[6] It is a *relation* and not a *function* as many kernel data structures in the executable side have many possible concrete representations.

Succinctly, the above statement can be read as: for related (Haskell and C) states s and s', such that $P\,s$ holds and $P'\ s'$ holds, the results of running a and c is related by return value–relation r, where the return value of the right-hand side is given by xf.

In this thesis, the details of what the return-value relation r and extraction function xf is not particularly important, so their definitions are always elided.

As an example, here is a correspondence rule we shall use later (Lemma B.3).

$$\frac{\text{ccorres}\ rv\ xf\ P\,C'\ hs\ a\ c}{\text{ccorres}\ rv\ xf\ (\lambda s.\ Q\,s \longrightarrow P\,s)\ P'\ hs\ (\text{stateAssert}\ \{\!|Q|\!\}\gg a)\ c}$$

This can be read as follows. Suppose we know that operations a and c correspond, given arbitrary preconditions P and C'. Now, suppose we prepended stateAssert $\{\!|Q|\!\}$ to the left-hand side, using the monadic bind operator $\gg$. Consider what would happen if we executed the left-hand side. If Q is false, then the left-hand side should fail, and our correspondence will vacuously hold. Otherwise, if Q is true, then we will proceed with a and have whatever effect a has. Knowing this, it must be that stateAssert $\{\!|Q|\!\} \gg a$ corresponds also to c, given that $Q\,s \longrightarrow P\,s$ for the initial state s. A key side-effect of this rule is that now that we have a precondition $Q\,s \longrightarrow P\,s$, we may use the fact that Q holds for the rest of the correspondence proof. I will elaborate on this soon.

2.7.3 Failure in the executable specification

You may notice that a correspondence statement assumes that its left-hand side, the Haskell code, does not fail. If the left-hand side were to fail, our statement is vacuously true! How can that be sound? The property that the Haskell functions do not fail is proved separately (as part of the refinement from the abstract specification to the executable), in the form of so-called "no-fail" lemmas. This alleviates the proof burden when we are proving correspondence. The proofs of non-failure are combined with correspondence at the end to obtain the overall refinement result.

There is a similar, related predicate called "empty-fail" lemmas. As the Haskell specification is embedded in a non-deterministic monad, each computation returns a *set* of possible next states. Empty-failure asks that if a function fails, then it returns *no* next states (i.e. it returns the empty set).

2.7.4 Composing correspondence statements

We have now seen the primary definitions used for correspondence proofs: the state relation between Haskell and C, the correspondence statement itself, and non-failure and empty-failure lemmas. In this section, we will discuss the proof techniques we use with these definitions, two main ways of composing correspondence statements, and how these proofs will be presented in this thesis.

Many rules about correspondence statements, such as the one presented above (Lemma B.3), can be applied in the usual natural deduction way. To do so, there is a crucial rule that we need: rules to *generalise* the preconditions of a correspondence statement[7]. We reproduce one such rule (Lemma B.2, the weaker variant).

$$\frac{\text{ccorres}\ r\ xf\ Q\ Q'\ hs\ f\ g \qquad \bigwedge s.\ A\ s \implies Q\ s \qquad \bigwedge s'.\ A'\ s \implies Q'\ s}{\text{ccorres}\ r\ xf\ A\ A'\ hs\ f\ g}$$

This rule (and its variants) is typically the first step of a proof. In doing so, the "current correspondence subgoal" (the leftmost premise of this rule) has arbitrary preconditions which we can instantiate however we like, with further rules. Most correspondence rules will have premises that take arbitrary preconditions, so this continues onwards until the proof is complete. This proof style allows us to more or less arbitrarily apply correspondence rules. Afterwards, we prove the second and third preconditions of the generalisation rule above, to show that those applications of correspondence rules were sound. For this reason, proofs are often made of two phases: a **correspondence step**, and a **preconditions step**. To illustrate this proof structure, let us take a look at an example.

Example 2.1 (Demonstration of composing preconditions)**.** Suppose we have the following two contrived rules:

$$\frac{\text{ccorres}\ r\ xf\ P\ P'\ hs\ a\ c}{\text{ccorres}\ r\ xf\ (\lambda s.\ B\ s \longrightarrow P\ s)\ P'\ hs\ a\ c}\ (1)$$

$$\frac{\text{ccorres}\ r\ xf\ P\ P'\ hs\ a\ c}{\text{ccorres}\ r\ xf\ (\lambda s.\ A\ s \longrightarrow P\ s)\ P'\ hs\ a\ c}\ (2)$$

for some fixed predicates A, A', B and B' (though P is still a variable predicate). Now, we wish to show

$$\frac{\text{ccorres}\ r\ xf\ C\ P'\ hs\ a\ c}{\text{ccorres}\ r\ xf\ A\ P'\ hs\ a\ c}$$

for another fixed predicate C. Suppose also that $\bigwedge s.\ A\ s \implies B\ s \implies C\ s$.

Proof. A formal Gentzen-style proof is given in Figure 2.1. However, providing a full proof tree for the later theorems in this thesis would be more confusing than helpful. Instead, we will present those proofs in the style as follows. Readers who have used Isabelle/Hol (or similar proof assistants) before may be familiar with this style, as I try to emulate Isabelle/Hol's "schematic variables" [Wenzel, 2014].

[7] This is conceptually similar to the "generalisation" or "consequence" rule for Hoare logic [Hoare, 1969].

Correspondence step. Our goal begins as

$$\text{ccorres}\ r\ xf\ ?_0\ P'\ hs\ f\ g\,.$$

Generalise our preconditions, as above.

First, apply rule (1). Our goal is now

$$\text{ccorres}\ r\ xf\ ?_1\ P'\ hs\ f\ g\,.$$

Next, apply rule (2). Our goal is now

$$\text{ccorres}\ r\ xf\ ?_2\ P'\ hs\ f\ g\,.$$

Lastly, apply our assumption (instantiating $?_2$ to C)!

Preconditions step. We instantiated our (executable) preconditions as follows. Note that we never instantiated the concrete preconditions to anything significant.

- $?_0 = (\lambda s.\ A \longrightarrow ?_1)$
- $?_1 = (\lambda s.\ B \longrightarrow ?_2)$
- $?_2 = C$

We call this a **precondition chain**. The precondition chain is terminated by C as the last step we make is to apply our precondition.

We have $?_0$ because we are given that $A \implies B \implies C$.

We have shown that the precondition chain holds, so we are done. □

The proof style used in this thesis "walks" up the branch of the proof tree that regards the correspondence statements. It makes explicit our choice of what rule to apply next with the variables of form $?_i$. This notation is inspired by and intended to correspond to *schematic variables* that Isabelle creates and instantiates as it proceeds through a proof script.

Note that it is idiomatic to try to avoid placing preconditions in the concrete precondition P' as much as possible. Although it is possible in the framework, in general we do not do have any proof obligations regarding the concrete preconditions.

The second technique for composing rules is the so-called "splitting"-style. Most correspondence statements for functions in the kernel have no premises (in the natural deduction sense). Instead they are between a Haskell function on the left, and either a Call statement or a literal operation on the right. To apply them, we must first adjust them into the right form. Applying them uses the so-called split rule, which has additional premises in terms of Hoare logic or the C-Simpl verification condition generator [Cock et al., 2008], that ask that the operations on either side preserve the preconditions. This process is supported by proof automation tactics. As we do not do many proofs in the splitting style in this thesis, I direct the reader to Winwood et al. [2009] for more details.

$$\dfrac{\dfrac{\dfrac{\text{ccorres}\ r\ xf\ C\ P'\ hs\ a\ c}{\text{ccorres}\ r\ xf\ (\lambda s.\ B\ s \longrightarrow C\ s)\ P'\ hs\ a\ c}\ (1)}{\text{ccorres}\ r\ xf\ (\lambda s.\ A\ s \longrightarrow (B\ s \longrightarrow C\ s)\ s)\ P'\ hs\ a\ c}\ (2) \qquad \dfrac{\vdots}{\bigwedge s.\ A\ s \Longrightarrow (A\ s \longrightarrow (B\ s \longrightarrow C\ s))\ s} \qquad \bigwedge s.\ P'\ s \Longrightarrow P'\ s}{\text{ccorres}\ r\ xf\ A\ P'\ hs\ a\ c}$$

Figure 2.1: A proof tree for Example 2.1.

2.8 Timing channels in data cache

Let us look at an example of a μTC, where a process infers information about another process through the execution time of an instruction. Many kinds of timing channels exist, but this kind will give us a stronger intuition for why Time Protection must partition memory.

Suppose we have a typical OS running two processes: a spy process and a victim process. The system is running on a single processor with a cache shared by all cores. The address spaces of both processes are mapped to disjoint pages, so neither process and read from addresses of the other process.

During the spy process' time slice, it carefully accesses specific memory addresses to overwrite every cache line with its own data, yielding to the victim process when it is done. The victim process runs, influencing the cache with each memory access and leaving a "footprint" in the cache. When control returns to the spy process, it accesses the same addresses it accessed before, carefully measuring the time it takes for each access. If an access was fast, the spy concludes there was a cache *hit*, and if an access was slow, the spy concludes there was a cache *miss*. For any cache misses, the spy can conclude that the victim process must have accessed a part of its memory that would write to that that cache line. Thus information has leaked from the victim process to the spy process, despite their memory having been partitioned, through a timing channel in the cache. This attack is known as a *prime and probe attack* [Ge et al., 2018].

2.9 How does time protection work?

To review, Time Protection eliminates timing channels between *domains* [Trustworthy Systems Team, 2017, 33]. As mentioned earlier in Section 2.3, a domain is a collection of threads that should be isolated from one another. The existing proof that the kernel enforces confidentiality give their strongest guarantees about domains [Murray et al., 2013], as we explain later in Section 2.10. If you want to limit the information one part of your system can observe about another, you should divide the threads of each part into separate domains.

The time protection mechanisms eliminate the timing channel in physically-addressed caches—as described earlier in Section 2.8—by partitioning the cache itself [Ge et al., 2019]. The OS allocates physical pages of memory in a careful fashion, such that each domain can only affect cache lines that no other domain can affect. It uses a technique called *page colouring*, which takes advantage of structure created by set-associative caches [Kessler and Hill, 1992]. Set-associative caches use certain *set-selector* or *index* bits of the memory address, to decide which section of the cache to store a value in. Each of these sections is disjoint, and is referred to as a cache "colour". On architectures where set-selector bits overlap with the page number of addresses, each page can only be resident in the section of cache of a particular colour. Therefore we can divide the colours, and hand each domain only the capabilities to pages of a particular set of colours. Then, threads in each domain can only map in pages of the chosen colours. With this mechanism enabled, a thread can access any virtual address (in *user-mode*),

and it will not touch cache lines that are not in the colour of its domain. Later, when we discuss domain "partitions", you can instead think "cache colour".

Since this mechanism relies on page allocation, it is limited to timing channels in physically-indexed caches. On most hardware, only off-core caches (generally L2…LLC) are physically indexed [Ge et al., 2019]. The timing channels in on-core caches must be handled differently; Wistoff et al. [2021] solves this with a hardware instruction to flush on-core state, during domain switches. Verification of those mechanisms does not depend on the outcomes of this project, so we consider timing channels in on-core caches not in the scope of our concerns.

It is important to note that the time protection implementation (and its verification explained below) is being done on an experimental modification of the seL4 microkernel that only supports a "separation kernel policy", which requires that *no* information is permitted to flow between domains. This policy is quite restrictive, and is only adopted in scenarios like a cloud hosting service, which would run VMs for mutually-distrusting clients that should not be able to interact with one another. In a cloud scenario, each VM would run in its own domain. Implementing and verifying time protection with less restrictive policies is left as future work for the overarching time protection project.

The first implementation of Time Protection was done by Ge et al. [2019], for the x86 architecture. Later, Buckley et al. [2023] re-implemented Time Protection for RISC-V, taking advantage of the `fence.t` instruction introduced by Wistoff et al. [2021, 2023]. This project builds the ideas of Buckley et al. [2023]. However, this project's proof artefacts are derived from the mainline seL4 verification work, because this project is independent of the implementation of Time Protection.

2.10 Verifying time protection enforcement

The partitioning described above is insufficient to show total assurance of our system. Note the caveat above that the partitioning of off-core caches is a kernel mechanism that only affects code run in user-mode. There are no mechanisms preventing the kernel, running in kernel-mode, from accessing any physical address it wishes to. As a result, a vulnerability could exist that would allow a malicious actor to craft system calls to read and write to a cache timing channel. No hardware mechanisms that can mitigate this exist. Software-level protection would have to trigger on every memory access, which would impose a sever performance penalty, so a software-level mechanism is also infeasible. In order to have total assurance of the security of our system, we must formally verify that calling the kernel will only access addresses in the partition of the currently-running domain.

There has been significant progress in verifying the time protection mechanisms for seL4. Sison et al. [2023] formalised the notion of a system being free of timing channel as follows. Timing channel freedom is expressed as a noninterference property [Goguen and Meseguer, 1982; Murray et al., 2013], for all pairs of domains D and D'. Informally speaking, *D does not interfere with D'* if for all sequences of system calls made by threads in any domain, the state of the OS is "observationally equivalent"—i.e. it looks the same—from the perspective of D',

with and without the effects of of system calls made by D. Timing channel freedom is defined by extending the notion of state to include microarchitectural state, modelled abstractly as

- the flushable state, such as on-core caches, branch-predictor state, etc. which can be cleared during domain switch time
- the portion of the partitionable state assigned to the current domain, such as the off-core cache lines of the same colour
- any parts of the partitionable state used for kernel shared data, and
- the exact wall-clock time.

Thus, if D does not interfere with D' with respect to this extended state, then we have eliminated the timing channel from D to D'.

Buckley et al. [2023] investigated timing channel-freedom for seL4 based on the above work by Sison et al. [2023]. Most relevant to our work is the following two contributions. They introduce the terminology of an **touched addresses** set (TA set), an over-approximation of the set of addresses that a system will access when run. More specifically it is the address of every kernel object the kernel has has retrieved since the last domain switch. Crucially, they also give an extension to the existing information flow proofs to cover timing channels, and hence show time protection enforcement. This extension requires provably correct tracking of the TA set, such that it obeys the invariant:

Proposition 2.2 (Partition subset invariant [Buckley et al., 2023, p.11])**.** *The physical translations of all addresses in the TA set form a subset of the union of the physical addresses that reside in the currently running user domain's partition.*

This project seeks to help obtain a proof of the partition subset invariant holds about the C implementation of seL4.

Verifying time protection enforcement hinges on a way to correctly track the TA set, and is an open research problem. We need the TA set so that we can reason about how the kernel affects off-core caches, which is partitionable state. Again, we need that the kernel only touches the partition of the cache that the currently running domain is permitted to touch. We shall see a method to track the TA set for the abstract specification by [Buckley et al., 2023] in Section 2.11.1

2.11 Related work

Lastly, we shall discuss some closely-related existing work, that I do not apply directly in this thesis. We shall see the method of reasoning about touched addresses put forward by Buckley et al. [2023], about whether there are any insights to be gained from verification projects for kernels other than seL4, and lastly some recent work on tracking touched addresses for machine code.

2.11.1 Tracking the touched addresses set for the abstract specification

The authors of Buckley et al. [2023] laid the groundwork to show time protection enforcement at the abstract specification level, and to use refinement to carry that property to apply to the C implementation. This direction is natural, as all previous security proofs are carried down by refinement from proofs over the abstract specification. Their approach to track the TA set was as follows. They add the TA set to the abstract state, to record what memory addresses have been accessed so far in execution. Note that the field is *ghost state*, which means it only exists in the specification of the kernel in the theorem prover, and does not exist in the C implementation at all. Before every access to a kernel object, they insert a statement to add that object to the TA set. Hence, at the end of a system call, the TA set field of the state contains every memory address that was accessed by the kernel. The authors add an assertion in the machine semantics for reading and writing to a memory address, that checks if the address is contained in the TA set. If the address is not contained, the access fails. Then, if the refinement proofs succeed, we can be certain that none of the above assertions are triggered, so all necessary objects must have been accounted for in the TA set.

However, this approach has met substantial proof engineering challenges. Many lemmas involved in the functional correctness proofs rely on the property that certain operations are *pure*, that is, they do not mutate state. By inserting these operations that add to the TA set, previously pure functions become impure, even though the state they are mutating is not real. Hence, large swathes of the refinement proofs are broken, and are yet to be repaired. Moreover, the above approach has a pitfall: At the abstract level, the layouts of all kernel objects have not been determined yet, and they only exist at lower steps of the refinement chain. This means that the TA set *grows* as we go down the specifications.

Suppose that we repair the refinement using the above approach, how our tracking of the TA set at abstract level could be refined down to the executable or concrete levels, such that the proof of time protection enforcement could also be refined, is still an open question. Our approach in this thesis circumvents this problem, and is described in Section 3.2.

2.11.2 Verifying isolation of an operating system with refinement

One of the earliest attempts at verifying isolation properties for an OS is KIT [Bevier, 1989]. They show that a machine-code implementation of the kernel is correct with regards to a specification, giving a precise semantics of the machine-code instructions. Their proofs are also done in a similarly compositional way: the author shows refinement of individual components of the kernel, before composing them to state refinement of the entire kernel. Their specification is quite simple, and the scale of their refinement is much smaller than that of seL4. Moreover, there is nothing that resembles tracking of touched addresses.

A detailed survey of OS verification projects is given by [Klein, 2009].

2.11.3 Capturing touched addresses for assembly verification

Verbeek et al. [2020] presents an automated method for verifying the memory usage of assembly code. Their process begins by using (unverified) tools to lift up assembly into a higher-level representation. Then, they find bounds on the touched addresses for each block of code by symbolic execution. They import these bounds into Isabelle/HOL to verify that they hold. Verification is done using novel Hoare logic that accounts for the memory addresses accessed by the program in each Hoare triple. Thus they use automated theorem proving techniques to prove the generated Hoare triples. Composing the Hoare triples allow the proofs to scale to cover large programs.

They define a memory usage Hoare triple as follows:

Definition 2.3 (Memory usage WRT a certain state change [Verbeek et al., 2020, p. 103])**.** The set of memory regions M is the *memory usage* WRT the state change from σ to σ', iff any byte at an address a not inside one of the regions is unchanged.

$$\text{usage}(M, \sigma, \sigma') \equiv \forall a.(\forall r \in M.[a, 1] \bowtie r) \implies \sigma' : *[a, 1] = \sigma : *[a, 1]$$

Here, σ and σ' refer to states of the program, the notation $[a, s]$ denotes a region of s bytes starting from memory address a, the relation $r_0 \bowtie r_1$ denotes that two regions do not overlap, and $*[a, s]$ denotes reading s bytes from address a. Intuitively, it says that, M is the only memory usage to get from a state σ to σ', iff the value at any address a that is *not* in M, is the same in both σ and σ'.

Definition 2.4 (Memory usage Hoare triple [Verbeek et al., 2020, p. 104])**.** A memory usage Hoare triple is defined as:

$$\{P\}\ f\ \{Q; M\} \equiv \forall \sigma\sigma'.P(\sigma) \wedge \text{exec_scf}(f, \sigma, \sigma') \rightarrow Q(\sigma') \wedge \text{usage}(M, \sigma, \sigma')$$

Assume exec_scf just means to take a “step” in the simulation of the program.

At first glance, using a Hoare logic for memory access seems highly promising, due to the ability to automatically prove Hoare triples, and that Hoare triples enable straightforward composition of proofs about subprograms. Moreover, the existing verification of seL4 already expresses properties using a Hoare logic. Perhaps we could adapt the current Hoare logic for seL4 to account for memory usage as well.

However, their Hoare logic has a fatal flaw: it cannot account for reads. As code that only reads from an address will not mutate it, that address can be excluded from the memory usage set, and the Hoare triple continues to hold. Reads are only accounted for insofar as the tool that does symbolic execution of code blocks will include them, but they are not part of the verification. For our purposes, since reads still impact the cache, we must provably account for them as well. It is unclear how to adapt their Hoare logic to account for reads as well.

Moreover, since their work is focused on machine code, many of their concerns are irrelevant to us. For example, they generate preconditions that the return address of a function is not

clobbered during the execution of a function body. Since we are dealing with C programs, that is not something that concerns us. The other side of our work being concerned with a C implementation, is that the C code we analyse will still be changed by compiler optimisations and instruction reordering. Considering how my chosen approach might be relevant for time protection elimination of the binary is out of scope.

2.12 Summary

In this chapter, we first covered key concepts of the formally-verified seL4 microkernel: its structure, kernel objects, and interface [Trustworthy Systems Team, 2017], the structure of the verification of seL4's functional correctness [Klein, 2009]; and saw an overview seL4's confidentiality proofs, which Time Protection aims to extend [Murray, 2013].

We saw that the Haskell implementation of seL4 is deeply embedded in a monadic style with its own Hoare logic [Cock et al., 2008], and how the C code is translated into C-Simpl and equipped with a model of variables [Winwood et al., 2009] and the heap [Tuch et al., 2007]. We discussed the refinement from seL4's Haskell implementation to the C implementation: the key definitions used, how correspondence statements are stated, and also composed [Winwood et al., 2009].

Next, we reviewed the notion of a μTC, prime-and-probe as an example of a μTC attack, and how the Time Protection mechanisms work to eliminate them [Ge et al., 2019].

Lastly, we reviewed some past work, from tracking the touched addresses at the abstract level [Buckley et al., 2023], past verification of operating systems [Bevier, 1989; Klein, 2009], and a similar attempt at tracking touched addresses for binaries [Verbeek et al., 2020].

Chapter 3

Formalised accounting of touched addresses

In this chapter, I shall

- Outline the ideal outcomes of my project, enriched by the context given in the previous chapter (Section 3.1),
- Describe how we will reason about the touched addresses set, particularly how we differ from Buckley et al. [2023] (Section 3.2),
- Explain how that approach embeds into the concrete specification (Section 3.3),
- Explain how that embedding can be verified via refinement from the executable specification (Section 3.4), and lastly,
- Justify the subset of the kernel I will demonstrate these methods on (Section 3.5).

3.1 Restatement of the goals of this project

Using what we have seen in the previous chapter, let us review the goals of this project. To verify Time Protection, we need a proof of the partition subset invariant (Proposition 2.2). This project will try to show that the invariant holds about seL4's *concrete specification*, as a *refinement* of the *executable specification*. To do so, I will extend the existing refinement to also account for touched addresses, in a way that is scalable to the entire kernel.

First, I define a way of encoding touched addresses for the executable specification and concrete specification. Then, I will develop methods for refining information about touched addresses at the executable level to the concrete level. I will demonstrate these methods on a carefully-chosen subset of the kernel, as a proof-of-concept that they may be applied for the entire kernel.

Let us clarify the scope of the project.

- We will assume certain invariants hold about the executable specification. How those assumptions will be eliminated is not in scope, and is the topic of another thesis project.
- We will find these results using interactive theorem proving. It is currently unclear how the problem may be reducible to a process that does not require human interaction (i.e. push-button verification). The refinement proofs we are extending are done interactively.
- Where relevant, our method does not need to support the full C language. Our work is for the purposes for verifying the seL4 microkernel, which has been written in a particular style of C [Klein et al., 2009].
- Lastly, like all previous time protection verification work, we assume a separation kernel policy and focus on the Risc-V version of seL4.[1]

In general, as formal verification is an expensive and tedious process, I will endeavour to minimise the work of a proof engineer that would be required to expand our method to the rest of the kernel.

3.2 Adding reasoning about touched addresses

In Section 2.10, we saw that the previous attempt at accounting for touched addresses by Buckley et al. [2023] faced difficulty. They encode the touched addresses as a set of addresses in kernel ghost state. Functions that access kernel objects first add the address of the object to the set. However, previously pure functions are now impure, invalidating large swathes of the proof-base which depended on purity.

There is an alternative approach that circumvents the issues encountered by Buckley et al. [2023]. This approach was proposed in private correspondence with my supervisors and other maintainers of the seL4 proofs. We believe that this approach is amenable to future verification of a multikernel version of seL4.

Rather than dynamically tracking the TA set itself during simulation, we instead show that each memory access is within the static partition of the currently running domain. This is a set that does not change at runtime, circumventing issues of making operations impure. The purpose is that the set of addresses in the current domain's partition *is* an over-approximation of the TA set, and obviously obeys the partition subset invariant (Proposition 2.2). We can encode that property by similarly inserting assertions at every memory access, that we shall

[1] Whether this assumption has significant implications on the refinement has not been explored deeply, as none of our work directly references any architecture-specific parts of the kernel. We choose Risc-V as Wistoff et al. [2021] implements the crucial `fence.t` instruction for Risc-V.

call *partition guards*.[2] With this approach, we do not modify any state, which avoids breaking existing proofs that slowed down the first attempt. But, because memory layouts of kernel objects are not determined until concrete specification, these assertions must live at concrete specification. In other words, we can no longer refine the proof of time protection enforcement from abstract specification down to concrete specification, and instead have to verify the time protection mechanisms directly at the concrete specification level.

Adding assertions at pointer accesses at the concrete specification has precedent. In order to adhere to the C standard, all pointer accesses are already surrounded by a guard that checks the address is not-null and aligned. The translation validation work, done by Sewell et al. [2013], strengthened all pointer dereference guards to check that the object being pointed to is of the correct type.

Not refining time protection enforcement seems to run counter to the precedent set by existing security proofs. However, we can still leverage refinement to carry useful invariants at the abstract specification level to show that our earlier assertions hold at the level of the concrete specification.

Finally, we arrive at where my project connects with earlier work. We need to verify that partition guards inserted at the concrete specification hold, judiciously assuming some invariants to be refined down from abstract specification. The assertions allow us to prove that the set of addresses accessed by the kernel—by virtue of these partition guards asserting that they are in the current domain's partition—obeys the partition subset invariant. The partition can then be used with the framework of Buckley et al. [2023] to produce a proof of time protection enforcement, as required. How the invariants will eventually be proven is the topic of Nair [2025], another thesis project.

To do so, we will insert partition guards at every pointer access. We will also assume that some useful invariants can be refined down from abstract specification to be used at concrete specification. If we can prove that the partition guards never trigger, using refinement from executable specification, we have proven the property we want.

From here on, where I say "verify" some part of the kernel, I mean to repair the refinement proof from the corresponding part of the executable specification.

3.3 Inserting the partition guards into the concrete specification

First, I shall discuss how I plan on implementing the partition guards, which is composed of two sub-problems: how to modify the concrete specification, and how the assertions can be stated formally.

[2] *Partition* is a slightly loaded term, as seL4's domains were previously called partitions [Murray et al., 2013]. Do not think "domain partitions" means "partition partitions"! A more precise term might be "domain partition guards" and "domain partition oracles". Nevertheless, we shall call them partition guards for the sake of brevity.

We can leverage existing infrastructure for inserting the partition guards. As mentioned above, when the concrete specification is derived, every pointer access is surrounded by a guard that checks the address is not-null and aligned. Sewell et al. [2013] developed substitution machinery to create a copy of the concrete specification with every pointer guard replaced with guards that give stronger guarantees about the object that is being pointed to. Then, a largely automatic proof shows that this new copy of the concrete specification is a refinement of the original. I will reuse the substitution machinery developed by the authors to insert partition guards for my project. This machinery is implemented in over 300 lines of Isabelle/ML code, so reusing it saves a considerable amount of time.

The much more difficult problem is how to state the assertions at all. Recall that cache colours are allocated to domains by the initial task, the thread that runs when the OS boots. In other words, the partition of the currently running domain is a function of the runtime state of the OS. In seL4, that manifests as the allocation of capabilities and what permissions those capabilities have to different capability spaces. Deriving properties from runtime security policy has been done already, as part of the work that showed seL4's access control mechanisms enforce integrity and authority confinement [Sewell et al., 2011]. There, the authors develop a model of security policy based on the capabilities that are present.[3] The proofs of integrity and authority confinement are expressed using this model, and they are trivially preserved by refinement to the concrete specification. This model can also be used to find the mapping of domains to partitions of colours as well. A complication arises because the above model is defined in terms of the abstract specification, but we wish to use this model at the concrete specification in order to phrase our assertions. How to obtain the partitioning at the abstract specification level and then refine it down to be used at the concrete specification level is an open research question, and the topic of another thesis project.

My project addresses this by making the first of two assumptions:

Proposition 3.1 (The Partition Oracle)**.** *There is a mapping of addresses to the domain partition they belong to (given by an* oracle *function).*

We can parametrise the refinement from executable specification to concrete specification over the partition oracle. One nice property of doing so is that if the oracle is set to be the trivial mapping where all domains are permitted to touch all memory addresses, then the assertions are trivially discharged. In this case, the partition guarded–concrete specification becomes semantically equal to the original specification.

3.4 Refining from the executable specification

Having answered how we can represent the partition subset invariant for the concrete specification, we now need to do the same for the executable specification.

[3]This is in a session called `Access`.

One useful property of the correspondence statements is that it assumes that the left-hand side—the Haskell operation—does not fail. As a corollary, any assertions that are in the left-hand side become additional preconditions which we can use to prove the correspondence. What assertions in the executable specification would help us discharge the partition guards in the concrete specification?

We can exploit the fact that seL4 is written in a very particular style. One rule it obeys is that it never takes pointers to the stack. In other words, each time the C implementation dereferences an address, it is the address of data on the heap, namely kernel objects! All the partition guards are surrounding accesses to kernel objects.

We would like to express the invariant that all accesses to kernel objects in the executable specification also only lie in the partition of the current domain. A natural approach is to mimic what we have done for the concrete specification and insert partition guards into the executable specification around each access to a kernel object. Let us call these **executable partition guards**, and the former ones **concrete partition guards**.

This leads us to our second and last assumption.

Proposition 3.2 (Structural similarity of kernel object accesses)**.** *Each pair of corresponding functions in the executable and concrete specifications perform accesses to the same kernel objects, in the same order.*

This assumption is motivated by how the existing refinement from executable to concrete heavily exploit their structural similarity Winwood et al. [2009]. Although this proposition is not strictly necessary for our approach to succeed, it makes our proofs easier and increases our confidence that our approach can be scaled to the whole kernel. We will see the importance of this conjecture during the demonstration of these methods in Chapter 4, particularly because of how we depend on one crucial rule Theorem 4.9.

How do we insert partition guards into the executable specification? There is no existing tooling for performing substitutions in the executable specification as there is for the concrete specification. Fortunately, in the Haskell implementation, all[4] kernel objects are instances of the `PSpaceStorable` typeclass, with methods

```
getObject :: PSpaceStorable a => PPtr a -> Kernel a
setObject :: PSpaceStorable a => PPtr a -> a -> Kernel ()
```

All accesses to kernel objects pass through these functions. Haskell has no pointers, but the executable specification simulates them by storing kernel objects in a global map of physical addresses to kernel objects. Every access of a kernel object in the concrete specification should have a corresponding call to `getObject` or `setObject`. Thus a simple way of inserting partition guards would be to update the definitions of `getObject` and `setObject` to have them. Doing so requires very little effort by the proof engineer, and also allows us to iteratively update the specification.

[4] All objects that are stored in physical memory.

I note that overriding `getObject` and `setObject` is not exhaustive. These two methods are operations of the `PSpaceStorable` typeclass. Kernel objects that are not only accessed by the kernel but also accessed by the hardware, namely virtual memory pages and hardware page tables, will override their definitions of `getObject` and `setObject`. However, these objects are exceptional because they affect other parts of the hardware. Our project is only concerned with memory accesses made by the kernel, so these are out-of-scope.

Importantly, both the executable and concrete specifications only model accesses to the *beginning* of kernel objects. An operation in the C implementation to set a single field of a struct is translated as overwriting the entire struct with one field changed. The Haskell implementation assumes that all kernel object accesses are aligned. Therefore, we proceed with the assumption that every dereference of a pointer by the concrete specification will have a corresponding dereference in the executable specification.

To summarise, we will insert similar partition guards into the executable specification. I believe this is sufficient to discharge the concrete partition guards,[5] because most functions are structurally similar, access the same objects in the same order, and all dereferences are aligned to the beginning of kernel objects. Additionally, doing so is simple, only requiring small modifications in a compositional way. Discharging the partition guards in the executable specification is not in scope Nair [2025].

3.5 Choosing a subset of the kernel

How can we choose a suitable representative fragment of the kernel to demonstrate our method on? One approach would be to verify a single system call, and growing our proof one system call at a time. To reduce the difficulty of a proof, we can sculpt the state of the kernel in the preconditions so that the system call triggers only one specific code path. However, even the simplest system calls pass through a large number of parts of the kernel. I believe verifying an entire code path within one year is likely unfeasible.

Alternatively, we can select individual functions of the kernel to prove our property about. Focusing on a single module of the kernel could enable more proof reuse. For example, the same invariants and other lemmas about a data structure in a kernel object could be used in multiple proofs about one module of code. Moreover, when we are selecting parts of the kernel to verify, we can select it at the granularity of functions or even sections of lines of code. Such flexibility is an asset under the limited time constraints of an Honours project. Thus, we will proceed with the latter approach.

In consultation with proof maintainers and kernel developers, I have also identified some criteria for selecting parts of the kernel.

- Prefer functions that do not depend on other system (i.e. "leaf" functions)

[5] Recall that discharging the executable guards is done by refinement from the abstract to executable specifications. Repairing that refinement is the goal of another project Nair [2025].

- Prefer functions that do not involve retyping memory.
- Prefer functions that modify minimal shared kernel data.
- Prefer functions that do not change the current running thread.

We avoid functions that involve retyping memory as they are already treated as an edge case in the existing proofs. They are not really key to the argument that such a proof method is viable for the entire kernel.

Global data structures that are shared between domains are an exception to the other partitioning of kernel objects. They are treated in a particular way [Ge et al., 2019] which makes them effectively uncoloured. As a result, accesses to shared kernel data that disobey the partition subset invariant are irrelevant. Moreover, there is active development on minimising the amount of shared kernel data, so what is and is not shared is in flux. It is possible that what we dismiss now as shared, could become partitioned in the future and must be accounted for.

We also avoid functions that change what the current running thread is. The reason is the same reason we avoid verifying complete syscalls: IPC interacts with too many subsystems of the kernel

In consultation with other proof and kernel maintainers, I settled on the function `lookupCap` and its dependencies.

3.6 Summary

In short, we represent the partition subset invariant by inserting partition guards into the executable and concrete specifications, surrounding each access to a kernel object. I give a brief argument why I believe that to be a sound representation, and why those partition guards in the executable specification is a sufficient assumption to resolve the concrete guards, at least for functions that are structurally similar (Proposition 3.2). Lastly, I describe our approach for choosing what part of the kernel to verify (`lookupCap`).

Chapter 4

Refinement of touched addresses on a simple part of the kernel

In the last chapter, we saw an approach to formalising the partition subset invariant for the concrete specification, and then the assumptions we need about the executable specification to prove it. In this chapter, I will describe the work that has been carried out this year. I apply the aforementioned methods on proving that the invariant holds for a set of simple procedures, `lookupCap` and `updateCap`, by repairing their correspondence proofs. However, the proofs require assumptions about a major dependency function `resolveAddressBits`. We will step through these proofs in sections Section 4.3 and Section 4.4. In this chapter, I refer to proving that the partition subset invariant holds for a function as "verifying" the function, or "repairing the correspondence for" the function.

The work was done on the verification of the Risc-V version of the kernel. Although I do not believe the correspondence proofs differ greatly based on the platform architecture, I did not investigate this in detail. As described in Chapter 3, the Risc-V version of the kernel has the strongest story for Time Protection, and is what the overall verification project is focusing on.

I have tried to simplify the notation in this chapter as much as possible for readability. Most of the Isabelle/Hol terms presented are not the same as the true terms in the proof script. I shorten the long noisy generated names for variables (and their projection and update functions) and also elides tedious coercions between logically equivalent types. Where there are significant deviations, they are marked, sometimes by a footnote.

4.1 Definitions and assumptions

First, I will cover how exactly I defined the partition oracle and guards. The partition guards in the executable specification are the primary assumption for our proofs.

4.1.1 The partition oracle

The partition oracle defined in terms of an underdefined function.

Definition 4.1 (The partition oracle)**.** Given a (virtual) memory address $a :$ machine-word, the domain that it belongs to is given by dom-of-addr a where

$$\text{dom-of-addr} \ : \ \text{machine-word} \rightarrow \text{domain}\,.$$

The partition of a domain $d :$ domain is given by partition-of-domain d where

$$\text{partition-of-domain}\ d \equiv \{\, a \mid \text{dom-of-addr}\ a = d \,\}\,.$$

Note that the oracle is explicitly *not* parametrised by the state of the kernel. On principle, the partitioning of memory between domains is a function of runtime state, because it is derived from the allocation of capabilities to the threads in each domain. Moreover, the time protection extension [Buckley et al., 2023] expects the touched addresses set as a function of state. However, the allocation of capabilities between domains for Time Protection is fixed once the system has booted [Ge et al., 2019]. I exploit this fact to simplify the proofs. Otherwise, we would have an additional proof burden of showing the state has not changed, each time we reference the partition oracle. How to reconcile this assumption with the reality that it is runtime state is left as future work (See Section 6.1).

By defining the oracle as a set comprehension, we get this injectivity property for free.

Lemma 4.2 (Domain partitions do not overlap)**.**

$$a \in \text{partition-of-domain}\ d \wedge a \in \text{partition-of-domain}\ d' \implies d = d'$$

Proof. If $a \in$ partition-of-domain d, then we have that dom-of-addr $a = d$. Similarly, we also show that dom-of-addr $a = d'$. It must be that $d = d'$. □

These definitions reflect the corresponding definitions in the proof artefacts of Buckley et al. [2023].

We will define some useful lemmas about partition oracle once we define the partition guards.

4.1.2 Partition guards in the concrete specification

As mentioned in Section 3.3, I reuse the substitution machinery from the translation validation work Sewell et al. [2013] to insert the partition guards for the concrete specification. The embedding used by the concrete specification is described in Section 3.3.

Definition 4.3 (Concrete specification–partition guard)**.** We will use the abbreviation

$$\text{dompart-guard}\ p\ s = p \in \text{partition-of-domain}\ (\text{ucast}\ (\text{ksCurDomain}\ (\text{globals}\ s)))\,.$$

The function ksCurDomain gets the currently-running domain from the global kernel state. In the concrete specification, domains are represented by 64-bit integers. However, in the executable specification, domains are represented by 8-bit integers, so we must cast it. So far, this cast has not caused any additional proof difficulty.

We use the substitution machinery to search the translated C code for statements of the form

$$\text{Guard C_Guard } \{s \mid \text{c-guard } (p\ s)\}\ b$$

and rewrite them with a partition guard[1] as

$$\begin{aligned}&\text{Guard C_Guard } \{s \mid \text{c-guard } (p\ s)\}\\ &\quad(\text{Guard C_Guard } \{s \mid \text{dompart-guard } (p\ s)\ s\}\ b)\,.\end{aligned}$$

Here, p is a pointer that is about to be dereferenced (in b), and b is the guard body that is executed if the guard holds. In the concrete specification, local variables in C are encoded as fields of a record, so p is a projection of the state record. Notice that we nest the guard statements within each other: the b on the first line has been replaced with another Guard term. We could have instead rewritten the predicate as the conjunction: $\{\, s \mid \text{ c_guard } p \wedge \text{dompart-guard } p\ s\,\}$. As we shall see later, the former representation makes it easier to "discharge" one guard at a time, because of existing tactics on correspondence statements.

Let us take a look at an example. Here is an excerpt from the C function `lookupCap`.

```
lookupCap_ret_t lookupCap(tcb_t *thread, cptr_t cPtr)
{
    lookupSlot_raw_ret_t lu_ret;
    lookupCap_ret_t ret;
    ⋮
    ret.cap = lu_ret.slot->cap;
    return ret;
}
```

The function `lookupCap` returns the variable `ret` which has type `lookupCap_ret_t`. The field `ret.cap` is initialised by *dereferencing* the pointer `lu_ret.slot`! In the concrete specification, prior to inserting the partition guards, the dereference is embedded as

$$\text{Guard C_Guard } \{\!|\, \text{c-guard } \acute{}\texttt{lu_ret.slot} \,|\!\}\ (\acute{}\texttt{ret.cap} :== \mathscr{H}\&(\texttt{lu_ret.slot} \rightarrow [\texttt{cap}]))\,.$$

After inserting the partition guards, we get the statement

$$\begin{aligned}&\text{Guard C_Guard } \{\!|\, \text{c-guard } \acute{}\texttt{lu_ret.slot} \,|\!\}\\ &\quad(\text{Guard C_Guard } \{s \mid \texttt{lu_ret.slot} \in \text{partition-of-domain } s\}\\ &\qquad(\acute{}\texttt{ret.cap} :== \mathscr{H}\&(\texttt{lu_ret.slot} \rightarrow [\texttt{cap}])))\,.\end{aligned}$$

[1]In reality, the machinery also inserts a guard that is useful for translation validation, but I omit those guards for brevity.

Thus, every pointer dereference has a guard asserting the pointer is in the partition of the current domain. We trust that the substitution procedure is correct and finds all the pointer dereferences in the concrete specification. Insertion of the partition guards is done *automatically*.

Correspondence statements are generally phrased as between (the translations of) a Haskell function and C function, by their names. After the insertion of these guards, the C function unfolds differently, and any existing correspondence proof will no longer succeed. The validity of statements between Haskell operations and non-Call SIMPL statements is unaffected.

I regret that the substitution has been done *in-place*, which has made our proof method less expressive. I will elaborate on this in Section 6.1.

4.1.3 Partition guards in the executable specification

In this section, we will see many definitions from the executable specification, which is derived from a Haskell implementation of the kernel. Although we will do proofs on their embedding inside the logic, I present them here as their Haskell code. Where the definition only exists inside the logic, I typeset them in the usual mathematical way.

First, we need to define a guard function, much like the concrete specification's dompart-guard.

Definition 4.4 (Executable specification–partition guard)**.** In Haskell, we define a partition guard function

```
isInDomainPartition :: Domain -> Word -> Bool
isInDomainPartition _ _ = True
```

However, we override that definition in the logic to be

$$\mathsf{isInDomainPartition}\ d\ p \equiv p \in \mathsf{partition\text{-}of\text{-}domain}\ d.$$

To keep the Haskell source actually runnable, and not needing to expose a partition oracle, the Haskell definition is a constant function that always returns `True`. In hindsight, it may have been more convenient if `isInDomainPartition` retrieved the current domain itself. However, whether that would have incurred additional proof burdens is unknown.

As mentioned in Section 3.4, the executable specification models pointer accesses with a global map from physical addresses to kernel objects. Specifically, every time the executable specification wishes to fetch or update a kernel object, it calls getObject and setObject with the physical address of that object.

To add partition guards for every pointer access, we create **partition guard–variants** of these functions in terms of the above guard function.

Definition 4.5 (Partition guard–variants for fetching and updating objects)**.** We define new Haskell functions as follows, in terms of the regular functions. (The final argument

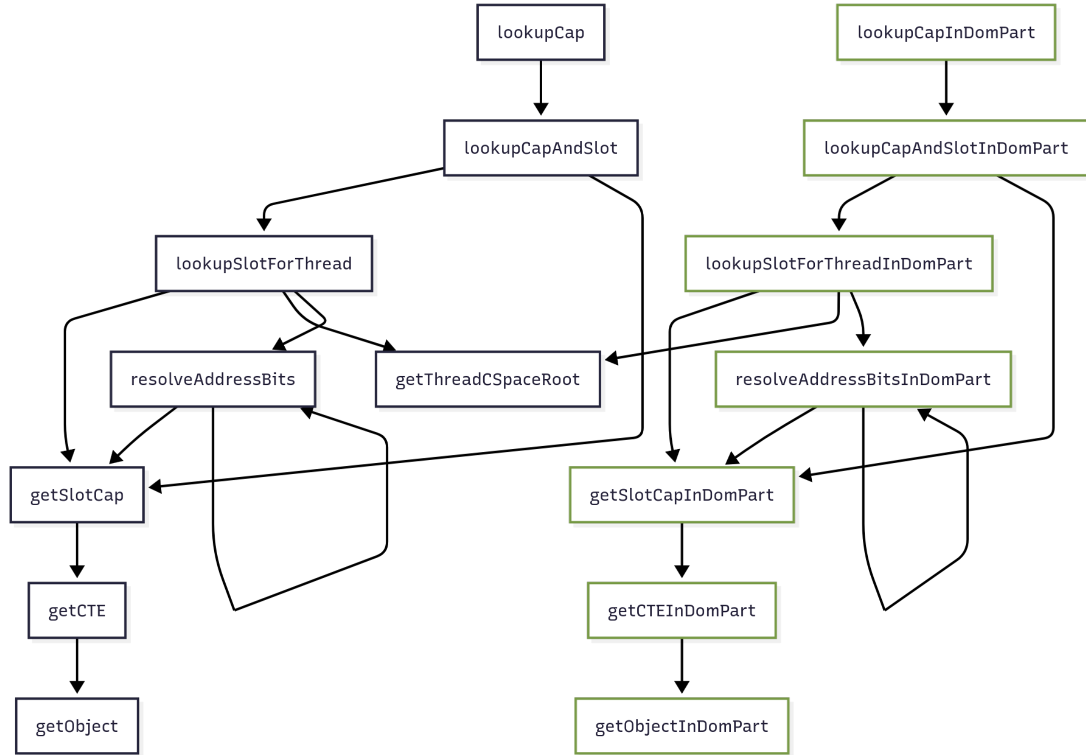

Figure 4.1: The new call-tree for partition-guard variants. On the left, in black, are the original functions. On the right, in green, are the partition guarded–variants. Notice that as `getThreadCSpaceRoot` does not access any kernel objects, it does not have a variant.

to `stateAssert` is the error message shown when the assertion fails, if the specification were executed.)

```
getObjectInDomPart :: PSpaceStorable a => PPtr a -> Kernel a
getObjectInDomPart ptr = do
        stateAssert (\s -> isInDomainPartition (ksCurDomain s) ptr)
            "object␣not␣in␣partition␣of␣current␣domain"
        getObject ptr

setObjectInDomPart :: PSpaceStorable a => PPtr a -> a -> Kernel ()
setObjectInDomPart ptr val = do
        stateAssert (\s -> isInDomainPartition (ksCurDomain s) ptr)
            "object␣not␣in␣partition␣of␣current␣domain"
        setObject ptr val
```

Then, as we choose to repair the correspondence statement of a particular function, we create a copy of its definition in the Haskell implementation, except we manually update any functions it calls to point to the guarded `getObjectInDomPart` and `setObjectInDomPart` instead. The new call tree is shown in Figure 4.1.

For example, consider the function `lookupSlotForThread`. It has the following definition.

```
lookupSlotForThread :: PPtr TCB -> CPtr -> KernelF LookupFailure (PPtr CTE)
lookupSlotForThread thread capptr = do
  threadRootSlot <- withoutFailure $ getThreadCSpaceRoot thread
  threadRoot <- withoutFailure $ getSlotCap threadRootSlot
  let bits = finiteBitSize $ fromCPtr capptr
  (s, _) <- resolveAddressBits threadRoot capptr bits
  return s
```

To define a version equipped with partition guards, we need to replace each call to a function that accesses kernel object, (i.e. `getSlotCap` and `resolveAddressBits`) with their partition guard–variants. The naming convention for functions with partition guards is to add the suffix `InDomPart`. So our new definition must call the functions `getSlotCapInDomPart` and `resolveAddressBitsInDomPart` respectively. We modify the above definition for the guarded variant `lookupSlotForThreadInDomPart` as follows.

```
lookupSlotForThreadInDomPart :: PPtr TCB -> CPtr ->
  KernelF LookupFailure (PPtr CTE)
lookupSlotForThreadInDomPart thread capptr = do
  threadRootSlot <- withoutFailure $ getThreadCSpaceRoot thread
  threadRoot <- withoutFailure $ getSlotCapInDomPart threadRootSlot
  let bits = finiteBitSize $ fromCPtr capptr
  (s, _) <- resolveAddressBitsInDomPart threadRoot capptr bits
  return s
```

The process of doing so is rather mechanical, and can likely be automated. I will elaborate on this in Section 6.1.

Mechanical details of the proof script structure Why not modify the definitions of `getObject` and `setObject` in-place, rather than create new definitions? There are a large number of invariants stated about definitions in the Haskell implementation. Notably, each definition often has accompanying **specification lemmas**, which are Hoare triples about its behaviour and that are vital for verification. An example of a specification lemma is given by Figure 4.2. The preconditions of these lemmas do not have enough information to discharge partition guards, so the operation would fail, and the lemmas would be false. To fix that, we would need to add preconditions about touched addresses to all of these specification lemmas, which would be messy. The new definitions with partition guards need specification lemmas as well, and it is often easier to prove these specification lemmas *using* the specification lemmas of the definition without the guards. Much like the guards in the concrete specification, it would be best to have these guarded variants in a new Isabelle/HOL locale (See Section 6.1).

$$\{\!|\ \text{valid-objs}'\ |\!\}\ \text{getSlotCap}\ t\ \{\!|\ \lambda r.\ \text{valid-cap}'r\ \text{and}\ \text{cte-at}'\ t\ |\!\}$$

Figure 4.2: A specification lemma for `getSlotCap`. The definitions of the predicates in the preconditions and post-conditions is not crucial.

4.2 An introduction rule for partition guards

Having defined the partition oracle and partition guards, we may now begin to prove theorems about them. We begin with a useful lemma.

Lemma 4.6 (Predicate for moving concrete specification–guards)**.**

$$\forall s\ s'.\ (s, s') \in \text{rf-sr} \wedge \text{isInDomainPartition}\ (\text{ksCurDomain}\ s)\ ptr \wedge \text{True} \longrightarrow \text{dompart-guard}\ ptr\ s$$

This lemma connects executable guards—which use isInDomainPartition—with concrete guard statements—which use dompart-guard. The rf-sr term is the *state relation*, which holds when the states of the executable and concrete specifications s and s' are equivalent.

Proof. Immediate from unfolding the definitions of isInDomainPartition and dompart-guard. Because the two states s and s' are related, both states must have the same currently-running domains, so the partitioning is the same. ☐

We need this oddly-shaped lemma to instantiate existing rules for moving guards in the concrete specification into preconditions of a correspondence statement. Below are rules that were proved in earlier work.

Lemma 4.7 (Move concrete specification–guards into preconditions)**.** *Rewrite a correspondence statement with a* Guard *term removed, given that the preconditions imply the guard's predicate.*

$$\frac{\forall s.\ s'.\ (s, s') \in sr \wedge P\,s \wedge P'\ s' \longrightarrow G'\ s' \qquad \text{ccorres}\ sr\ xf\ \ A\,C'\ \ hs\,a\,c}{\text{ccorres}\ sr\ xf\ \ (A\ \text{and}\ P)\ (C' \cap \{\,s \mid P'\ s\,\})\ hs\ a\ (\text{Guard}\ F\{s \mid G'\ s\}\ c)}$$

$$\frac{\forall s.\ s'.\ (s, s') \in sr \wedge P\,s \wedge P'\ s' \longrightarrow G'\ s' \qquad \text{ccorres}\ sr\ xf\ \ A\,C'\ \ hs\,a\,(c\ ;;\ d)}{\text{ccorres}\ sr\ xf\ \ (A\ \text{and}\ P)\ (C' \cap \{\,s \mid P'\ s\,\})\ hs\ a\ ((\text{Guard}\ F\{s \mid G'\ s\}\ c)\ ;;\ d)}$$

If we apply these rules in a backwards-reasoning style, we "eliminate" the Guard *term.*

What is the difference between the two rules? The first rule shows correspondence of

$$a \qquad \text{and} \qquad \text{Guard } F\{\, s \mid G' s \,\}\, c$$

whereas the second rule shows correspondence between

$$(a \qquad \text{and} \qquad \text{Guard } F\{\, s \mid G' s \,\}\, c) \;;;\; d\,.$$

When might we use the second rule instead of the first? Having the tail operation d is slightly more convenient, as it allows this rule to be applied as an introduction rule while we are in the middle of a proof. We can only apply the former rule as an introduction rule, if it is the last rule of a correspondence proof. Also note that because the executable specification is embedded in a monadic style, it is impossible to determine whether a term a is a single statement or several statements bound. We still have to express the left-hand side as the arbitrary a, making the left and right-hand sides have different shapes. The choice of rule is, to my knowledge, a matter of convenience.

If we instantiate the leftmost premise of these rules with Lemma 4.6, we get these two useful rules which let us convert concrete specification–partition guards on the right-hand side, into preconditions in terms of isInDomainPartition.

Lemma 4.8 (Move concrete specification–partition guards)**.** *Rewrite a partition guard on the right-hand side as a precondition.*

$$\frac{\text{ccorres } sr\ xf\ A\ C'\ hs\ a\ c}{\begin{array}{c}\text{ccorres } sr\ xf\ (A \text{ and } (\lambda s.\ \text{isInDomainPartition (ksCurDomain } s)\ p))\ C'\ hs \\ a\ (\text{Guard C_Guard}\ \{\, s \mid \text{dompart-guard } p\ s \,\}\ c)\end{array}}$$

$$\frac{\text{ccorres } sr\ xf\ A\ C'\ hs\ a\ (c \;;;\; d)}{\begin{array}{c}\text{ccorres } sr\ xf\ (A \text{ and } (\lambda s.\ \text{isInDomainPartition (ksCurDomain } s)\ p))\ C'\ hs \\ a\ ((\text{Guard C_Guard}\ \{\, s \mid \text{dompart-guard } p\ s \,\}\ c) \;;;\; d)\end{array}}$$

Proof. By Lemma 4.7, where the first precondition is discharged by Lemma 4.6, and taking P to be

$$\lambda s.\ \text{isInDomainPartition (ksCurDomain } s)\ p.$$

□

These rules will allow us to prove an *introduction rule* for partition guards.

Theorem 4.9 (Partition guard–introduction)**.** *Given an arbitrary correspondence statement between operations a and c ;; d, we may prepend a partition guard to both the left and right-hand side operations.*

$$\frac{\text{ccorres } r\ xf\ P\ P'\ hs\ a\ (c\ ;;\ d)}{\begin{array}{c}\text{ccorres } r\ xf\ (\lambda s.\ \text{isInDomainPartition}\ (\text{ksCurDomain}\ s)\ p \to P\ s)\ (P' \cap \{\, s \mid p = p' \,\})\ hs \\ (\text{stateAssert}\ (\lambda s.\ \text{isInDomainPartition}\ (\text{ksCurDomain}\ s)\ p) \gg a) \\ ((\text{Guard C_Guard}\ \{\, s \mid \text{dompart-guard}\ p'\ s \,\}\ c)\ ;;\ d)\end{array}}$$

One could imagine a rule between just a and a program c. As we will not apply this rule in this thesis, we elide it.

Before we prove it, we ought to first understand each part of it. Let us start from the bottom last parameter of the conclusion.

We prepend a partition guard to the left-hand side, connected with the ≫ operator (monadic bind that ignores the parameter). Then, we surround the right-hand side with a partition guard.

Entirely for convenience, the precondition of the concrete state requires $p = p'$, as pointer expressions have different syntactic representations in the concrete and executable specifications. You could imagine an alternate rule where $p = p'$ is its own premise; although equally expressive, the current presentation is slightly more convenient when using existing tactics for correspondence proofs.

Lastly, the precondition on the executable state is that for some state s

$$\text{isInDomainPartition}\ (\text{ksCurDomain}\ s)\ p \longrightarrow P\ s.$$

This is not an additional proof obligation on the user of the rule, but actually makes the rule more universally applicable. (Where usually the precondition would just be P, now P is in in *negative* position.) Recall that at the end of a correspondence proof, we must show that the preconditions of each correspondence statement we have used implies the next. As a result of the precondition, after the introduction rule is applied, we permanently have the fact isInDomainPartition (ksCurDomain s) p in that precondition proof. This is helpful when there is more than one partition guard on a single value between both sides (which occurs in the proof in Section 4.4). In that case, you can apply the introduction rule once, then apply the regular guard moving rules multiple times. The preconditions generated by the guard moving–rules can be discharged by the precondition obtained from the introduction rule.

Let us prove this rule.

Proof. We proceed in a backwards-reasoning style and begin by generalising the preconditions with Lemma B.2. Because the concrete precondition references both p and p', which appear in each of the left and right-hand sides, we need the states to be related to relate p and p'. Hence we will use the "stronger" variant in this proof.

Our goal begins as

$$\begin{array}{c}\text{ccorres } sr\ xf\ ?_0\ P'\ hs \\ (\text{stateAssert}\ (\lambda s.\ \text{isInDomainPartition}\ (\text{ksCurDomain}\ s)\ p) \gg a) \\ (\text{Guard C_Guard}\ \{\, s \mid \text{dompart-guard}\ p\ s \,\}\ c)\end{array}$$

where variables of the form $?_i$ represent unknown parts of the goal that will be filled in as we apply rules.

Correspondence step. Fix our executable and concrete states as s and s'. Next we take cases on whether the predicate isInDomainPartition (ksCurDomain s) p holds, which is the predicate of the executable partition guard we are introducing. Suppose it is the case that it does not hold. Then the left-hand side operation must fail, as the call to stateAssert will trigger. Thus, our correspondence is vacuously true and we can then assume we are in the other case. In other words, we assume that

$$\text{isInDomainPartition}\ (\text{ksCurDomain}\ s)\ p\,. \tag{4.1}$$

Formally, this step can be done by applying Lemma B.1. I elide the proof of the empty-fail $(\cdots)$ premise as it is uninteresting, and can found automatically by the verification condition generator proof automation developed by Cock et al. [2008]. Our goal is now:

$$\begin{gathered}\text{ccorres}\ sr\ xf\ ?_1\ P'\ hs\\ (\text{stateAssert}\ (\lambda s.\ \text{isInDomainPartition}\ (\text{ksCurDomain}\ s)\ p) \gg a)\ c^{\cdot}\end{gathered}$$

To remove the right-hand side guard, apply Lemma 4.8, making our goal:

$$\begin{gathered}\text{ccorres}\ sr\ xf\ ?_2\ P'\ hs\\ (\text{stateAssert}\ (\lambda s.\ \text{isInDomainPartition}\ (\text{ksCurDomain}\ s)\ p) \gg a)\ c^{\cdot}\end{gathered}$$

Next, to remove the left-hand side guard, apply Lemma B.3. That gives the goal:

$$\text{ccorres}\ sr\ xf\ ?_3\ P'\ hs\ a\ c\,.$$

This is our assumption—the arbitrary correspondence statement—so we are done with this step.

Preconditions step. We applied three correspondence rules, and instantiated our (executable) preconditions as follows. As usual, we never instantiated the concrete preconditions to anything significant.

- $?_0 = (\lambda s.\ \text{isInDomainPartition}\ (\text{ksCurDomain}\ s)\ p \longrightarrow ?_1)$
- $?_1 = ?_2$ and $(\lambda s.\ \text{isInDomainPartition}\ (\text{ksCurDomain}\ s)\ p)$
- $?_2 = (\lambda s.\ \text{isInDomainPartition}\ (\text{ksCurDomain}\ s)\ p \longrightarrow ?_3)$
- $?_3 = (\lambda s.\ \text{isInDomainPartition}\ (\text{ksCurDomain}\ s)\ p \longrightarrow P)$

The precondition chain is terminated by P as the last step we make is to use the arbitrary correspondence statement in our assumptions, which has the (executable) precondition P.

We must show $?_0$, which holds by transitivity of implication.

We have instantiated correspondence statements for the entire left and right-hand side of our goal, and shown that each precondition implies the next. Our proof is complete. □

Thus, we have proved a crucial rule:

- Lemma 4.7, which lets us rewrite concrete partition guards as preconditions, and
- Theorem 4.9, which lets us introduce partition guards to both sides of a correspondence statement, and adds that the guard holds to the precondition chain.

In particular, Theorem 4.9 is perhaps the most important theorem I present. We shall see how it is applied in the following section. The reason we need Proposition 3.2 is that this rule is easiest to apply for structurally similar procedures.

4.3 Repairing correspondence of a simple procedure

Equipped with an introduction rule for partition guards and other useful lemmas, we can finally verify some programs. We will

- Explore how to phrase refinement between guarded-variants of functions, and then define and prove a key so-called "upgrading lemma" (Section 4.3.1)
- Discuss temporary assumptions we make as the proof has been left to later work (Sections 4.3.1 and 4.3.2), and then
- Wrap up the proof (Section 4.3.4).

In the preceding sections, we set the stage for being able to refine touched addresses from the executable to concrete specifications. As mentioned in Chapter 3, I demonstrate the proposed methods by repairing the correspondence of the function `lookupCap`,[2] or in other words, that the Haskell function `lookupCapInDomPart` corresponds to the C function `lookupCap` with partition guards inserted. From here on however, although I refer to them as "functions", the sides of the correspondence statements that comprise the refinement proof are not necessarily C or Haskell functions. For example, `getSlotCap` corresponds to the C operation `slot->cap`, where `slot` has type `cte_t *`. Figure 4.3 shows the dependency tree between "functions", as well as which "functions" have correspondence statements phrased about them. In this section, we will generally refer to correspondence statements by the name of the function in the executable specification. The names in the executable specification can be considered the canonical ones.[3]

[2]Note that its name is the same in both the executable and the concrete specifications.

[3]In fact, the executable specification used to be known as the "design specification".

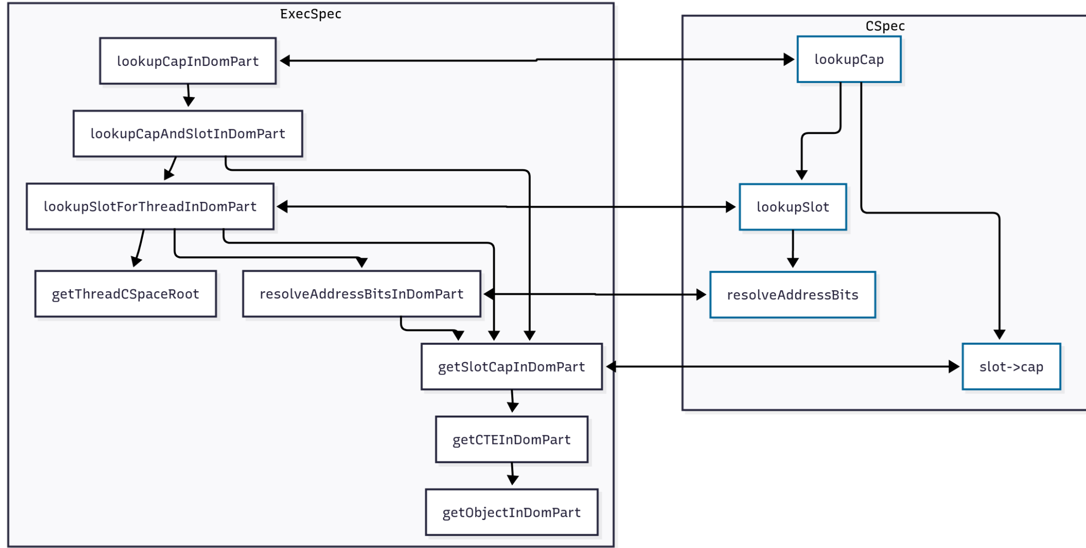

Figure 4.3: Operation dependency trees for the Haskell and C `lookupCap` functions. The horizontal arrows represent operations that have correspondence statements proved about them.

We will build up the repaired correspondence proof for `lookupCapInDomPart` bottom-up, but before that let us briefly discuss the structure of the repairs. Repairing correspondence of the top-level functions `lookupCapInDomPart` and `lookupSlotForThreadInDomPart` is straightforward. Both functions do not dereference any pointers themselves, so their correspondence proofs can be repaired by instead composing correspondence statements for the partition guard–variants. At the bottom of the diagram, the three Haskell functions `getSlotCapInDomPart`, `getCTEInDomPart`, `getObjectInDomPart` all correspond to one line of C code, `slot->cap`. This is the tricky part, and the part we will spend most of our time. Lastly, we will assume correspondence of `resolveAddressBitsInDomPart`. I will justify why I believe that making that assumption is reasonable later.

4.3.1 Extending correspondence to guarded variants

Let us begin with developing the correspondence proof of `getSlotCapInDomPart`. It helps to first review the correspondence statement of `getSlotCap`. (There is a subtle difference not shown in the diagram. The original proofs show correspondence between `getSlotCap` and the pointer dereference `slot->cap`.) Formally, we have the following.

Theorem 4.10 (Correspondence statement for getSlotCap). *A call* getSlotCap *ptr corresponds to dereferencing the value of ptr to access the field* cap.[4]

[4]Two technical details are omitted. As we note in Figure 4.2, the pointers on the left and right-hand sides have different syntactic representations. They require additional ceremony to be equated. Hidden by ellipsis are premises and preconditions that roughly state that *x* is not a global variable.

```
getSlotCapInDomPart :: PPtr CTE -> Kernel Capability
getSlotCapInDomPart ptr = do
    cte <- getCTEInDomPart ptr
    return $ cteCap cte

getCTEInDomPart :: PPtr CTE -> Kernel CTE
getCTEInDomPart = getObjectInDomPart
```

Figure 4.4: The Haskell definitions of `getSlotCapInDomPart` and `getCTEInDomPart`. The purpose of the alias `getCTEInDomPart` is to instantiate the type variable in the type of `getObjectInDomPart`.

$$\frac{\cdots}{\text{ccorres ccap-relation } xf \text{ True } \cdots \text{ (getSlotCap } p) \; (x \; :== \mathscr{H}\&(p \rightarrow [\text{cap}]))}$$

We want to show correspondence between `getSlotCapInDomPart` and the guarded pointer dereference, or more formally that for some set of premises, we have that:

$$\begin{aligned} &\text{ccorres } r \; xf \;\; P \, P' \;\; hs \\ &\quad (\text{getSlotCapInDomPart } ptr) \\ &\quad (\text{Guard C_Guard } \{s.ptr \in \text{partition-of-domain } s\} \; (x \; :== \mathscr{H}\&(p \rightarrow [\text{cap}]))) \, . \end{aligned}$$

We shall build up a proof of this.

First, the shape of our information about touched addresses is quite different between the left and right-hand sides.

The definitions of `getSlotCapInDomPart` and `getCTEInDomPart` is shown in Figure 4.4 On the left-hand side, the partition guard is buried at the bottom of the call tree, through a call to `getCTEInDomPart` and then to `getObjectInDomPart`. On the right-hand side, we already have the expression for the pointer dereference. Can we "lift" the guard on the side of the executable specification up the call tree, to the level of `getSlotCapInDomPart`?

Lemma 4.11 (Rewrite getSlotCapInDomPart in terms of stateAssert)**.** *A call to the function* getSlotCapInDomPart *is equivalent to a partition guard and then* getSlotCap.

$$\begin{aligned} &\text{getSlotCapInDomPart } ptr = \textbf{do} \\ &\quad \text{stateAssert } (\lambda s. \text{ isInDomainPartition (ksCurDomain } s) \; ptr); \\ &\quad \text{getSlotCap } p \\ &\; \textbf{od} \end{aligned}$$

Proof. Unfold the definitions of all the functions getSlotCap, getCTE, getSlotCapInDomPart, and getCTEInDomPart. The equality holds by associativity and the distributive property of monadic bind. □

We have one last useful lemma to define and prove. Since we still have the original correspondence statement of `getSlotCap`, can we use it to prove correspondence of the partition-guarded variant `getSlotCapInDomPart`? In a way, we "upgrade" the existing proof, so we call this kind of lemma an **upgrading lemma**.[5]

Lemma 4.12 (Upgrading lemma for getSlotCap)**.** *Given correspondence of* getSlotCap*, we have correspondence with the guarded variant* getSlotCapInDomPart *and a partition guard.*

$$\frac{\text{ccorres}\ r\ xf\ P\ P'\ \ hs\ (\text{getSlotCap}\,p \gg\!= f)\ (c\ ;;\ c'\ \ p)}{\begin{array}{c}\text{ccorres}\ r\ xf\ P\ P' \cap \{\, s \mid p = p' \,\}\ hs\ (\text{getSlotCapInDomPart}\,p \gg\!= f)\\ ((\text{Guard C_Guard}\ \{\, s \mid \text{dompart-guard}\ p'\ \ s \,\}\ c)\ ;;\ c'\ \ p)\end{array}}$$

Proof. We shall proceed in a backwards-reasoning style and generalise the preconditions with Lemma B.2.

Our goal begins as

$$\begin{array}{c}\text{ccorres}\ r\ xf\ ?_0\ Q'\ \ hs\ (\text{getSlotCapInDomPart}\,p \gg\!= f)\\ ((\text{Guard C_Guard}\ \{\, s \mid \text{dompart-guard}\ p'\ \ s \,\}\ c)\ ;;\ c'\ \ p)\,.\end{array}$$

Correspondence step. First, rewrite the conclusion using Lemma 4.11 to obtain

$$\begin{array}{c}\text{ccorres}\ r\ xf\ ?_0\ Q'\ \ hs\\ \left(\begin{array}{ll}\textbf{do} & \text{stateAssert}\ (\lambda s.\ \text{isInDomainPartition}\ (\text{ksCurDomain}\ s)\ ptr);\\ & \text{getSlotCap}\ ptr\\ & f\\ \textbf{od} & \end{array}\right)\\ ((\text{Guard C_Guard}\ \{\, s \mid \text{dompart-guard}\ p'\ \ s \,\}\ c)\ ;;\ c'\ \ p)\,.\end{array}$$

This matches the form of our introduction rule (Theorem 4.9)! Applying it gets us to

$$\begin{array}{c}\text{ccorres}\ r\ xf\ ?_1\ Q'\ \ hs\\ (\text{getSlotCap}\,p \gg\!= f)\\ ((\text{Guard C_Guard}\ \{\, s \mid \text{dompart-guard}\ p'\ \ s \,\}\ c)\ ;;\ c'\ \ p)\,.\end{array}$$

which is our assumption. We are done.

Preconditions step. The only correspondence statement we applied was Theorem 4.9, which leaves our preconditions untouched. There is nothing to prove.

Having shown correspondence of each part of the operations, we are done. ☐

[5] "Upgrade" is an arbitrarily-chosen synonym for the word "lift", which already carries too many meanings.

This upgrading lemma will be used later in the proof of Theorem 4.19, to rewrite a call to `getSlotCapInDomPart` with one to `getSlotCap`. As a proof of concept, we will now prove correspondence of `getSlotCapInDomPart`.

Theorem 4.13 (Correspondence of getSlotCapInDomPart). *A call* getSlotCapInDomPart *ptr corresponds to a partition guard on ptr, followed by dereferencing the value of ptr to access the field* cap.[6]

$$\frac{\dots}{\begin{array}{c}\text{ccorres}\ r\,x\,f\,P\,P' \cap \{\, s \mid p = p' \,\}\ hs\ (\text{getSlotCapInDomPart}\,p) \\ (\text{Guard C_Guard}\ \{\, s \mid \text{dompart-guard}\ p'\ s \,\}\ (x\ :==\ \mathscr{H}\,\&(ptr \rightarrow [\text{cap}])))\end{array}}$$

Proof. As this proof is not key to the overall argument, I will give only a proof sketch.

We can apply the rules Lemma B.4 and Lemma B.5 to append dummy statements: Skip for the right-hand side, and ≫=return for the left-hand side. Then, we can apply Lemma 4.12. That rule leaves our preconditions untouched, so there are no obligations regarding the preconditions either. We are done! □

Let us summarise the key points involved in completing this proof.

- Lift the partition guards on the left-hand side up to the level of the proof. We found a rule to rewrite getSlotCapInDomPart into a partition guard followed by a call to getSlotCap (Lemma 4.12).
- Then, by applying the partition guard–introduction rule Theorem 4.9 we encode the argument that the left-hand side *must* fail, and so we may assume it does not.
- That gives us enough information to discharge the the guard on the right-hand side.

This proof forms the primary technical contribution of my project. We will revisit this proof later and explore its ramifications, but to summarise: We have composed both existing lemmas and newly-defined lemmas to use only the *knowledge that partition guards on the left-hand side must succeed*, in order to discharge partition guards on the right-hand side. The new correspondence statements do not have any new preconditions; they are identical to the old statements, except that the left-hand sides now contain (and assume the success of) partition guards.

[6] As in Theorem 4.10, the premises are hidden by ellipsis. The premises are similar to those in elided in Theorem 4.10: They are just what is convenient to apply the theorem later and are not crucial for the proof.

4.3.2 An additional assumption

The major caveat in my proof of correspondence for `lookupCap` is that we assume the correspondence for the function `resolveAddressBitsInDomPart`.

Proposition 4.14 (Correspondence of resolveAddressBitsInDomPart)**.** *We elide difficult parts of the statement with ellipsis, as this statement is not proven and so they are unimportant.*

$$\begin{gathered}\text{ccorres } (\cdots) \text{ rab-xf } (\cdots)\ (\cdots)\ [\,] \\ (\text{resolveAddressBitsInDomPart } \mathit{cap}'\ \mathit{cptr}'\ \mathit{guard}') \\ (\text{Call } \texttt{resolveAddressBits})\end{gathered}$$

Due to the difficulty of this proof, it is left as future work. See Section 6.1. However, this assumption is *not used* for the proof of correspondence of `getSlotCapInDomPart` above (Section 4.3.1). To clarify, it does *not* take away from the primary technical contribution of this project.

Similarly, we also assume its specification lemma. The purpose of this is described in Section 4.3.3.

Proposition 4.15 (Specification lemma of resolveAddressBitsInDomPart)**.**

$$\begin{gathered}\{\!\!\{ \text{valid-objs}' \text{ and valid-cap}'\ \mathit{cap} \}\!\!\} \\ \text{resolveAddressBitsInDomPart } \mathit{cap}\ \mathit{addr}\ \mathit{depth} \\ \{\!\!\{ \lambda rv.\ \text{cte-}\mathit{at}'\ (\text{fst } rv) \}\!\!\}\end{gathered}$$

The specifics of what the predicates (valid-objs′, valid-cap′, etc.) are defined to be is not critical to my argument.

4.3.3 Extending specification lemmas for guarded variants

When we prove correspondence for `lookupSlotForThread` and `lookupCap`, we do so in the "splitting"-style that generates subgoals about whether the functions called uphold the correct postconditions. These subgoals are resolved automatically by the verification condition generator and weakest precondition proof automation [Cock et al., 2008; Schirmer, 2006; Winwood et al., 2009]. However, we need to supply them the same *specification lemmas* about the partition guarded–variants (`getSlotCapInDomPart` and `lookupSlotForThreadInDomPart`) that exist for the regular functions (`getSlotCap` and `lookupSlotForThread`).

Lemma 4.16 (Calling getSlotCapInDomPart does not mutate the state)**.**

$$\{\!\!\{ P \}\!\!\} \text{ getSlotCapInDomPart } t\ \{\!\!\{ \lambda rv.\ P \}\!\!\}$$

Lemma 4.17 (Specification lemma for `getSlotCapInDomPart`)**.** *This is nearly identical to the specification lemma shown in Figure 4.2.*

$$\{\!\!\{ \text{valid-objs}' \}\!\!\} \text{ getSlotCapInDomPart } t\ \{\!\!\{ \lambda r.\ \text{valid-cap}'\ r \text{ and cte-at}'\ t \}\!\!\}$$

Lemma 4.18 (Specification lemma for `lookupSlotForThreadInDomPart`)**.**

$$\{\!\mid \text{valid-objs}' \mid\!\}\ \text{lookupSlotForThreadInDomPart}\ t\ addr\ \{\!\mid \lambda r.\ \text{cte-at}'\ r \mid\!\}$$

Proof. The proofs of all three lemmas proceed the same. First, we rewrite in terms of stateAssert with the corresponding lemma (Lemma 4.11). Then, by unfolding definitions, we arrive at what we need. In Isabelle, it is done by the existing weakest precondition proof automation [Cock et al., 2008], which draws on the above specification lemma Proposition 4.15. □

4.3.4 Finishing the top-level proof

We have shown correspondence for `getSlotCap`, and the specification lemmas required to verify correspondence of `lookupSlotForThread` and then `lookupCap`. The proofs here are done by amending the existing proof of correspondence for the functions without partition guards. A thorough presentation of those proofs would detract from the clarity of this manuscript; I direct the interested reader to [Winwood et al., 2009] and to the proof artefacts.

Theorem 4.19 (Correspondence of lookupSlotForThreadInDomPart)**.**

$$\begin{gathered}\text{ccorres}\ (\cdots)\ (\cdots)\ (\text{valid-pspace}'\ \text{and}\ \text{tcb-at}'\ \mathit{thread})\ (\cdots)\ []\\ (\text{lookupSlotForThreadInDomPart}\ \mathit{thread}\ \mathit{cptr})\\ (\text{Call}\ \texttt{lookupSlot})\end{gathered}$$

Proof. Proceed the same as the original proof, renaming any references to the called functions (e.g. getSlotCap) with the new partition guarded–variant (e.g. getSlotCapInDomPart).

At calls to getSlotCapInDomPart, rewrite them with Lemma 4.12.

When lookupSlotForThreadInDomPart finally calls resolveAddressBitsInDomPart, apply Proposition 4.14 and Proposition 4.15 to discharge that part of the correspondence. □

And at last, our top-level result.

Theorem 4.20 (Correspondence of lookupCap)**.**

$$\begin{gathered}\text{ccorres}\ (\cdots)\ (\cdots)\ (\text{valid-pspace}'\ \text{and}\ \text{tcb-at}'\ a)\ (\cdots)\ []\\ (\text{lookupCapInDomPart}\ a\ b)\\ (\text{Call}\ \texttt{lookupCap})\end{gathered}$$

Proof. Same as above. □

I omit the exact return-value relations and extraction functions used, as they are unimportant.

Importantly, we have *not added any new preconditions* to these correspondence statements compared to the versions without partition guards. We are able to discharge the concrete guards entirely by using the information obtained from the executable guards!

```
updateCapInDomPart :: PPtr CTE -> Capability -> Kernel ()
updateCapInDomPart slot newCap = do
        cte <- getCTEInDomPart slot
        setCTEInDomPart slot (cte { cteCap = newCap })
```

Figure 4.5: The definition of `updateCapInDomPart`.

```
cap_untyped_cap_set_capFreeIndex(cap_t cap, uint64_t v64) {
    cap.words[1] &= ~0xfffffffffe000000ull;
    cap.words[1] |= (v64 << 25) & 0xfffffffffe000000ull;
    return cap;
}
```

Figure 4.6: The definition of `cap_untyped_cap_set_capFreeIndex`.

4.4 Testing the methods on another procedure

A crucial goal of my project is to show that my methods are scalable to the entire kernel. In this section, we shall see the ideas from the previous section applied to repairing the correspondence proofs of an independent but related function `updateCap`. As before, we want to show correspondence from the partition guarded–variant `updateCapInDomPart`, whose definition is shown in Figure 4.5.

The function `updateCap` is quite similar to `lookupCap`, as it interacts with no complicated data structures and is a straight-line function. All it does is write a capability object into a capability table entry (CTE).

However, unlike `lookupCap`, it corresponds to *two* operations in the C implementation. It corresponds to a single line of C code that sets a capability through a pointer: `*dest = cap`. However, it also has a correspondence statement with `cap_untyped_cap_set_capFreeIndex`, whose definition is shown in Figure 4.6. This sets the "free index" of a capability, which involves careful bitwise operations to the words that the capability composed of.

The function `updateCapInDomPart` in the executable specification is used in two correspondence statements with different concrete operations. Currently, only one of those statements has been repaired, and the other is incomplete. I proved correspondence between `updateCapInDomPart` and the operation `*slot = cap`. The correspondence statement between `updateCapInDomPart` and the function `cap_untyped_cap_set_capFreeIndex` is incomplete, but nearly done.

First, let us take a look at how we can prove the former statement. Then I will discuss the difficulties with proving the latter.

We want to prove[7]

$$\text{ccorres dc xfdc} \top (\cdots)\ hs\ (\text{updateCapInDompart}\ dest\ cap)\ (\mathcal{H}(\&(dest \rightarrow \texttt{cap}) \mapsto cap))\,.$$

Following the same structure as the correspondence proof for `getSlotCap`, we want to rewrite `updateCapInDomPart` in terms of `stateAssert`. Although there are two accesses to a kernel object and thus two partition guards, logically they are guarding the same pointer, *slot*. That leads us to this lemma.

Lemma 4.21 (Rewrite updateCapInDomPart in terms of stateAssert)**.**

$$\begin{aligned}&\text{updateCapInDomPart}\ dest\ cap = \textbf{do}\\ &\quad \text{stateAssert}\ (\lambda s.\ \text{isInDomainPartition}\ (\text{ksCurDomain}\ s)\ dest);\\ &\quad \text{updateCap}\ dest\ cap\\ &\ \textbf{od}\end{aligned}$$

Proof. This proof proceeds identically to Lemma 4.11, except that after unfolding, we have two partition guards on the left-hand side. However, two guards checking the same condition is idempotent, so we may rewrite it with just one. □

Proceeding the same as the proof of Theorem 4.10, we will prove an upgrading lemma for `updateCap`.

Lemma 4.22 (Upgrading lemma for updateCapInDomPart)**.** *Given correspondence of the base* updateCap*, we have correspondence with the guarded variant* updateCapInDomPart *and a partition guard.*

$$\frac{\text{ccorres}\ r\ xf\ P\ P'\ \ hs\ (\text{updateCapInDomPart}\ dest\ cap \gg\!= f)\ (c\ ;;\ c'\ \ p)}{\begin{array}{c}\text{ccorres}\ r\ xf\ P\ P' \cap \{\, s \mid p = p' \,\}\ hs\ (\text{updateCapInDomPart}\ dest\ cap \gg\!= f\,)\\ ((\text{Guard C_Guard}\ \{\, s \mid \text{dompart-guard}\ p'\ \ s \,\}\ c)\ ;;\ c'\ \ p)\end{array}}$$

Proof. We can prove this the same way we proved Lemma 4.12; it is immediate after applying the rewrite rule Lemma 4.21 and then the introduction rule Theorem 4.9. □

Now we can prove the overall correspondence statement.

Theorem 4.23 (Correspondence statement for updateCapInDomPart)**.**

$$\text{ccorres dc xfdc} \top (\cdots)\ hs\ (\text{updateCapInDomPart}\ dest\ cap)\ (\mathcal{H}(\&(dest \rightarrow \texttt{cap}) \mapsto cap))\,.$$

Proof. Identical to Theorem 4.10. Hooray! □

[7] I elide the definitions of dc and xfdc as they are not important to the proof.

The similarity in the proof structure suggests that these proofs can be automated, but we leave that for future work *Section* 6.1.

The second correspondence statement, with `cap_untyped_cap_set_capFreeIndex`, is in a broken state. However, I have made enough progress in the proof such that the partition guards do not occur in positive position anywhere—we do not have any obligations to *discharge* the partition guards, we just have them as (useless) assumptions! To speculate on why the proofs do not proceed, in the correspondence step of the original proof, most of the work is done by proof automation. However, in my modified proof, I had to manipulate the partition guards manually. It is likely that rearranged the subgoals and assumptions just enough that the existing proof does not succeed.

Why not define an upgrading lemma? A major flaw of the current proof technique is that, although the partition guards are inserted in the executable specification in new definitions (those with names ending in `InDomPart`), the partition guards are inserted *in-place* in the concrete specification. This means that very few correspondence statements still hold, and for any correspondence statement whose right-hand side is a Call statement, we cannot "lift" the guards up the way we do on the left-hand side. Concretely, we would like to rewrite a hypothetical Call `cap_untyped_cap_set_capFreeIndex_in_dom_part` as

Guard C_Guard (⋯) ;; Call `cap_untyped_cap_set_capFreeIndex`

but cannot yet do so. This is a shame, as there is an existing library of tools for rewriting C-Simpl expressions. I will elaborate on this in Section 6.1.

4.5 Summary

I sought to demonstrate how to track the touched addresses set, such that we can show that the partition subset invariant holds, in the way outlined by Chapter 3. We defined our partition oracle, and a method for inserting partition guards into both the executable and concrete specifications. Next, we repaired correspondence of `getSlotCap`, and managed to reuse those proof methods on a similar function `updateCap`. To do so, we first defined an introduction rule for partition guards (Theorem 4.9), proved rewrite rules to rewrite Haskell functions in terms of stateAssert, and then composed existing correspondence rules.

As a result of the proof, I show that partition guards in the executable specification are *sufficient* to discharge partition guards in the concrete specification; no other preconditions are necessary. We transform the knowledge that the executable guards must have succeeded to argue that the concrete guards must also. This validates our hope that, when we have structurally similar functions (Proposition 3.2), our limited assumptions about the executable specification are sufficient.

I also highlight some flaws with the implemented approach, which I elaborate on in Section 6.1.

Chapter 5

Discussion of scalability of the proposed proof methods

5.1 How much effort did these proofs take?

A natural metric to measure scalability is time spent, so-called "proof effort". Below I present breakdown of how the time was spent for each part of this demonstration. Looking at these numbers too closely can be a bit misleading. Each task was not done completed sequentially, and rather each required several iterations. Moreover, despite using the unit person-weeks, these are not weeks of full-time concentrated effort, but rather many development sessions interleaved with other coursework. The tasks are presented in rough chronological order—which differs slightly from the order the work has been presented—which means I was less familiar with the tooling and environment (e.g. using Isabelle, the structure of the seL4 proof base, and the refinement framework).

Task	Effort (person-weeks)
Setup partition oracle and guards	~4
Repair compositional correspondence lemmas	~6
Repair of `getSlotCap`	~5
Repair of `updateCap`	~1

The most remarkable trend in these figures is that repairing the correspondence statements of `updateCap` took only one week, while repairing the simpler function `getSlotCap` took more than a month. Although I regret that this is a small sample size, it suggests that it was not difficult to adapt the proof techniques for `getSlotCap` to `updateCap`. In other words, the techniques are applicable to other parts of the kernel.

However, the figures are strongly skewed because of the effort of becoming familiar with the tools and environment. It is safe to say it required several months for me to feel comfortable

with writing the proofs. Before that time, I received significant technical advice from my supervisor and assessor as well.

I note that Staples et al. [2014] conducted a survey on the verification of seL4, and demonstrate a strong correlation between proof effort and lines of proof. However, I believe that this project is a poor sample. While Staples et al. [2014] analyses work that is dominated by writing proofs, roughly one third of my project was spent amending the specification to reason about touched addresses. Moreover, rather than thousands of lines of function definitions and implementation, the total change I made to (the hand-written part of) the concrete specification is less than ten lines of code. Yet, doing so took about the same amount of time as the proof of correspondence for `getSlotCap`, a proof that involved many times more lines of code. Thus, the stark difference in project structure suggests that we should not expect the results of Staples et al. [2014] to be applicable.

Lastly, because I discovered this research late in the project, it was too late to properly measure the same factors. With that being said, if these methods are applied properly to the entire kernel, we ought to analyse whether they turned out as scalable as predicted. In that case, it would be crucial to apply the same productivity criteria. I briefly mention this again in (Section 6.1).

5.2 Do these techniques apply to other parts of the kernel?

So far, we have only had a glimpse of the procedures in the kernel. The functions `lookupCap` and `updateCap` do not contain any loops, or any recursion, nor do they interact with any complicated data structures. The one complicated structure we encountered is the CSpace trees in the definition of `resolveAddressBits`. However, we currently assume correspondence of `resolveAddressBits`. The simplicity of targeted functions is intentional; as outlined in Chapter 3, we stray away from tricky behaviour like kernel object retyping, inter-process communication, and so on. Due to the complexity that tricky components of the kernel impose on the functional correctness proofs, we can infer that they would bring their own problems for our goal of refining touched addresses.

A complex structure does not necessitate complexity when refining touched addresses, however. Complicated functions that compose many other functions, but themselves do not dereference any pointers do not need to be repaired, as no partition guards have been inserted. Generally, when we show correspondence of two functions, they are structurally similar (Proposition 3.2), and each line has a natural corresponding line on the other side. This is a property that was exploited for the original refinement proofs between executable and concrete specifications [Winwood et al., 2009]. In these cases, for each guard on the right-hand side, there should be another guard in the same place on the left-hand side. The correspondence between `updateCap` and `set_capFreeIndex` is an interesting case. On the left-hand side, we had two partition guards. On the right-hand side, as there are two array index operations, we have four partition guards, albeit they check that the same pointer is in the correct partition. In short, structural similarity between the executable and concrete sides make doing this proof simpler. We must

hope that most pointer dereferences will have corresponding guards on the executable side that can be used to discharge them.

5.3 Summary

I believe that the result of this project is sufficient to be cautiously optimistic that our methods are scalable to the entire kernel. The effort of each phase of this project is strongly skewed by my unfamiliarity with Isabelle, the seL4 proof structure, and particularly the correspondence framework. I believe that, at least for components of the kernel that are structurally similar across their Haskell and C implementations (Proposition 3.2), my approach will be sufficient and applicable.

Chapter 6

Conclusion

This thesis investigates how to extend seL4's refinement from its intermediate executable specification to its concrete specification, such that we can reason about the touched addresses of the kernel. The purpose of this reasoning is a future proof of Time Protection, where we must show the *partition subset invariant*: that the kernel does not dereference pointers that point outside the *partition of the currently-running domain* (Proposition 2.2).

It achieves this goal by describing how to encode touched addresses through the insertion of *partition guards* such that the partition subset invariant is obtained by refinement (Chapter 3), demonstrating how to extend the existing refinement proofs to cover the partition guarded–specifications (Chapter 4), and lastly constructing an argument that these methods are encapsulated and reusable for the rest of the kernel (Chapter 5).

I end this thesis by discussing some future work, and the implications of this thesis for the verification of Time Protection.

6.1 What comes next?

Throughout the thesis, I have alluded to possible improvements to the work I have presented. I have accumulated that discussion here.

Obviously, the natural goal is to extend this approach to cover the entire kernel, and repair the entire refinement proof—a far-away goal! What are some steps before we achieve that?

The immediate next step is to finish the proof of correspondence between `updateCap` and the helper function `cap_untyped_cap_set_capFreeIndex`. As mentioned in Chapter 4, the current breakage is unrelated to our touched addresses refinement, and it would be a crucial additional example that the methods are scalable.

A crucial assumption that I have made is the correspondence statement for and specification lemma for `resolveAddressBits`. I leave this for future work as it is a notoriously complicated

procedure. It traverses a CSpace object (a recursive data structure, roughly a tree of capability entries), and the existing correctness proof is remarkably complicated in comparison to its surrounding proofs. In the executable specification, it is defined as a recursive function, whereas in the concrete specification, it contains a while loop. It is difficult to connect these different control flow structures, as well as to set up the induction for the left-hand side and to state and prove the loop invariants for the right-hand side. However, this proof will eventually have to be done, but I believe it can be deferred until simpler parts of the kernel are repaired.

So far, all the repaired functions have been part of the "CNode module" of the kernel, and I believe pursuing the repair of that entire module next is a natural way forward. It would cause us to encounter our first true data structure, doubly-linked lists, as the kernel uses them to hold MDB nodes.

Another way to bolster the argument that these methods are scalable to the entire kernel would be to conduct a survey on the "complexity" of functions in seL4 (that have correspondence statements about them). This survey must ask questions such as:

- How often do functions dereference pointers?
- How often do functions just compose others?
- How often are the Haskell and C function structurally identical?

As we have seen, the answers to these questions is strongly correlated to the difficulty of repairing a correspondence proof.

Another priority should be a more rigorous analysis of the methods' scalability. We can repeat the experimental design of Staples et al. [2014], recording perceived effort and project-level factors such as schedule pressure.

Insert concrete partition guards into a new module A critical flaw of the current implementation of the approach, as mentioned at the end of Chapter 4, is that the partition guards are inserted into the concrete specification in-place. A proper application of this approach should use the substitution framework to create guarded-variants in a locale.[1] Doing so gives us the same expressiveness in the right-hand side of a correspondence statement, as we have had in the left-hand side. We could rewrite the right-hand sides of statements in terms of a Guard statement followed by the base operation. To repeat an example given in Chapter 4, we would like to rewrite a hypothetical Call `cap_untyped_cap_set_capFreeIndex_in_dom_part` as

Guard C_Guard (⋯) ;; Call `cap_untyped_cap_set_capFreeIndex`

but cannot yet do so. One question that needs to be answered is how existing lemmas about the unmodified C functions could be reused for the new modified C functions, or if that is possible at all.

[1] In Isabelle/HOL, locales roughly correspond to the notion of modules in general-purpose programming languages. See the section on locales in Wenzel [2014, p. 103] for the details.

Automatic insertion of executable partition guards In a similar vein, it would save proof engineering effort to extend the Haskell translator to also generate the partition guarded–variants. An additional benefit would be that these definitions do not need to clutter the Haskell source code. Because this project only required repairing the correspondence of a few functions, I deemed that implementing the tooling for generating these definitions was not worth it.

Lastly, to quicken the process of applying these methods to the rest of the kernel, it would be helpful to investigate more proof automation. So far, I only present some helpful lemmas, and some general guidance on doing proofs (rewrite as an assert, write an upgrading lemma, apply the introduction rule, etc.). As more of the proofs are repaired and patterns emerge, what can be automated will become more apparent.

After these methods have been applied to the entire kernel, the most urgent question is how to instantiate the Time Protection theory with them? At the moment, the partition subset invariant is embedded as the specifications equipped with assertions. However, it is not in a closed form or a proposition at all; we might like a *functional* form of the invariant. How this form can be obtained requires further investigation.

In the broader context of this project, we need a proof of the invariants we assume about the executable specification (i.e. the partition guards and oracle), from the abstract specification. That is the topic of another thesis project, Nair [2025], and will be completed in future work.

6.2 Closing remarks

The sel4 microkernel is poised to be the first OS microkernel with a formally-verified story for eliminating μTCs with its Time Protection mechanisms. As our computer systems are increasingly shared with untrustworthy users, and increasingly execute more untrustworthy software, eliminating μTCs is crucial for total assurance of security. This thesis forms a small first but optimistic step towards one phase of verifying Time Protection and more secure computer systems.

Bibliography

William R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.

Scott Buckley, Robert Sison, Nils Wistoff, Curtis Millar, Toby Murray, Gerwin Klein, and Gernot Heiser. Proving the absence of microarchitectural timing channels. *arXiv preprint arXiv:2310.17046*, 2023.

David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In Otmane Ait Mohamed, César Muñoz, Sofiène Tahar, editor, *International Conference on Theorem Proving in Higher Order Logics*, pages 167–182, Montreal, Canada, August 2008. Springer. doi: 10.1007/978-3-540-71067-7_16.

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

DoD 1986. *Trusted Computer System Evaluation Criteria.* Department of Defence, 1986. DoD 5200.28-STD.

Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8:1–27, April 2018. ISSN 2190-8508. doi: https://doi.org/10.1007/s13389-016-0141-6.

Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: the missing OS abstraction. In *EuroSys Conference*, Dresden, Germany, March 2019. ACM. doi: 10.1145/3302424.3303976.

Joseph Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, California, USA, April 1982. IEEE Computer Society.

Gernot Heiser. The seL4 microkernel – an introduction. seL4 Foundation Whitepaper, May 2020. URL https://trustworthy.systems/publications/papers/Heiser_20:sel4wp.abstract.pml.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969. ISSN 0001-0782. doi: 10.1145/363235.363259.

R. E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10:338–359, 1992.

Gerwin Klein. Operating system verification — an overview. *Sadhana*, 34(1):26–69, February 2009.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009. ACM.

Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Haburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwartz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy*, pages 19–37, San Francisco, CA, US, 2019.

Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16: 613–615, 1973. doi: 10.1145/362375.362389.

Moritz Lipp, Michael Schwartz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*, Baltimore, MD, USA, 2018.

Carl Metz and Nicole Perlroth. Researchers discover two major flaws in the world's computers. *The New York Times*, 2018. URL https://www.nytimes.com/2018/01/03/business/computer-flaws.html.

Robin Milner. An algebraic definition of simulation between programs. Technical Report 14, Computation Services Department, University College of Swansea, 1970.

Robin Milner. An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, IJCAI'71, page 481–489, San Francisco, CA, USA, 1971. Morgan Kaufmann Publishers Inc.

Toby Murray. On the limits of refinement-testing for model-checking CSP. *Formal Aspects of Computing*, 25(2):219–256, February 2013. doi: 10.1007/s00165-011-0183-6.

Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy*, pages 415–429, San Francisco, CA, May 2013. IEEE. doi: 10.1109/SP.2013.35.

Sai Nair. Updating L4v invariants to aid time protection proofs. BSc(Hons) thesis, School of Computer Science and Engineering, Sydney, Australia, November 2025.

Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

Thomas Sewell. *Translation Validation for Verified, Efficient and Timely Operating Systems*. PhD thesis, UNSW, Sydney, Australia, July 2017.

Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. In Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editor, *International Conference on Interactive Theorem Proving*, pages 325–340, Nijmegen, The Netherlands, August 2011. Springer. doi: 10.1007/978-3-642-22863-6_24.

Thomas Sewell, Magnus Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 471–481, Seattle, Washington, USA, June 2013. ACM.

Robert Sison, Scott Buckley, Toby Murray, Gerwin Klein, and Gernot Heiser. Formalising the prevention of microarchitectural timing channels by operating systems. In *International Symposium on Formal Methods (FM)*, Lübeck, DE, March 2023. Springer. doi: 10.1007/978-3-031-27481-7_8.

Mark Staples, Ross Jeffery, June Andronick, Toby Murray, Gerwin Klein, and Rafal Kolanski. Productivity for proof engineering. In *Empirical Software Engineering and Measurement*, page 15, Turin, Italy, September 2014.

Data61 Trustworthy Systems Team. *seL4 Reference Manual, Version 7.0.0*, September 2017.

Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editor, *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 97–108, Nice, France, January 2007. ACM. ISBN 1-59593-575-4.

Freek Verbeek, Joshua A. Bockenek, and Binoy Ravindran. Highly automated formal proofs over memory usage of assembly code. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 98–117, Cham, 2020. Springer International Publishing. ISBN 978-3-030-45237-7.

Makarius Wenzel. *The Isabelle/Isar Reference Manual*, August 2014.

Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. Mind the gap: A verification framework for low-level C. In S. Berghofer, T. Nipkow, C. Urban, M. Wenzel, editor, *International Conference on Theorem Proving in Higher Order Logics*, pages 500–515, Munich, Germany, August 2009. Springer. doi: 10.1007/978-3-642-03359-9_34.

Nils Wistoff, Moritz Schneider, Frank Gürkaynak, Luca Benini, and Gernot Heiser. Microarchitectural timing channels and their prevention on an open-source 64-bit RISC-V core. In *Design, Automation and Test in Europe (DATE)*, virtual, February 2021. IEEE.

Nils Wistoff, Moritz Schneider, Frank Gürkaynak, Gernot Heiser, and Luca Benini. Systematic prevention of on-core timing channels by full temporal partitioning. *IEEE Transactions on Computers*, 72(5):1420–1430, 2023. doi: 10.1109/TC.2022.3212636.

Notation used in this thesis

The empty list is denoted with [], á la Haskell.

Throughout this document, you will see the notation

$$\{\!|\, P \,|\!\}$$

for some boolean proposition P. It is syntactic sugar for a *set comprehension*, that is

$$\{ x \mid P x \}$$

or the set of all states x such that $P\,x$. For our purposes, x will often be named s, as it refers to the state of the kernel.

This is commonly used with the kernel Hoare logic, as the preconditions and postconditions are predicates on the state: Hoare triples have the form

$$\{\!|\, P \,|\!\}\ f\ \{\!|\, \lambda rv.\ Q \,|\!\}.$$

Useful existing proved lemmas

Throughout this thesis, particularly in Chapter 4, we make use of many already-proved lemmas. To keep the prose in earlier chapters clear, but to still be exhaustive, I produce the less crucial lemmas here. I am unsure whether they are formally published anywhere.

Lemma B.1 (If the left-hand side would fail, assume it does not)**.** *This is the key rule that transforms assertions on the left hand side to satisfy preconditions.*

$$\frac{\{\!\!\{\lambda s.\ \neg P\, s\}\!\!\}\ a\ \{\!\!\{\mathsf{False}\}\!\!\} \qquad \mathsf{empty\text{-}fail}\ a \qquad \mathsf{ccorres}\ r\ xf\ G\, G'\ \ hs\ (a \gg\!= b)\ c}{\mathsf{ccorres}\ r\ xf\ (\lambda s.\ P\, s \longrightarrow G\, s)\ G'\ \ hs\ (a \gg\!= b)\ c}$$

Lemma B.2 (Generalisation of correspondence preconditions)**.** *This rule has both a weak variant, that is used often, and a stronger variant, which requires that the executable and concrete states be related.*

$$\frac{\mathsf{ccorres}\ r\ xf\ Q\, Q'\ \ hs\ f\, g \qquad \bigwedge s.\ A\ s \implies Q\, s \qquad \bigwedge s'.\ A'\ s \implies Q'\ s}{\mathsf{ccorres}\ r\ xf\ \ A\ A'\ \ hs\ f\, g}$$

$$\frac{\mathsf{ccorres}\ r\ xf\ Q\, Q'\ \ hs\ f\, g \qquad \begin{array}{c}\bigwedge s\, s'.\ A\ s \wedge s' \in A' \wedge (s, s') \in \mathsf{sr\text{-}rf} \implies Q\, s \\ \bigwedge s\, s'.\ A\ s \wedge s' \in A' \wedge (s, s') \in \mathsf{sr\text{-}rf} \implies s' \in Q'\end{array}}{\mathsf{ccorres}\ r\ xf\ \ A\ A'\ \ hs\ f\, g}$$

Lemma B.3 (A left-hand side assertion corresponds to doing nothing)**.** *If precondition P holds, then* stateAssert $\{\!\!\{\mathsf{P}\}\!\!\}$ *has no effect.*

$$\frac{\mathsf{ccorres}\ rv\ xf\ \ P\, C'\ \ hs\ a\, c}{\mathsf{ccorres}\ rv\ xf\ \ (\lambda s.\ Q\, s \longrightarrow P\, s)\ P'\ \ hs\ (\mathsf{stateAssert}\ \{\!\!\{Q\}\!\!\} \gg a)\ c}$$

Lemma B.4 (Remove a Skip at the end of the right-hand side)**.** *This rewrites the right-hand side to remove skip statements. If substituted in reverse, you can* insert *a skip statement.*

$$\mathsf{ccorres}\ \cdots\ a\, (c\ ;;\ \mathsf{Skip}) = \mathsf{ccorres}\ \cdots\ a\, c$$

Lemma B.5 (Remove a return at the end of the left-hand side)**.** *This rewrites the left-hand side to remove redundant* return. *If substituted in reverse, you can* insert *a* return.

$$\mathsf{ccorres}\ \cdots\ (a \gg\!= \mathsf{return})\ c = \mathsf{ccorres}\ \cdots\ a\, c$$