UNSW
AUSTRALIA

School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Updating L4v Invariants to Aid Time Protection Proofs

by

Sai Nair

Thesis submitted as a requirement for the degree of

Bachelor of Advanced Computer Science

Submitted: November 2025

Supervisor: Dr. Rob Sison

Student ID: z5418256

Abstract

Microarchitectural timing channels threaten the security of computers across the globe. Recently, a set of operating system mechanisms, collectively called *time protection*, was proposed by Qian Ge as a way to close these channels, and was implemented and demonstrated to work correctly in an experimental version of seL4. A follow-up paper by Buckley, Sison et al. described a potential workflow for formalising time protection in seL4 by linking these mechanisms to existing proofs about the flow of information between security domains.

This report discusses a modified approach to formalising time protection in seL4, and details the efforts made to add an invariant on top of seL4's abstract specification. The invariant specifies that any kernel objects accessible to a security domain may only reference other memory regions accessible to that domain. This report also discusses the process through which this result can be utilised in more concrete specifications of seL4 to show that it partitions kernel memory in accordance with time protection.

Acknowledgements

First and foremost, I would like to thank Dr. Rob Sison and Dr. Thomas Sewell for their guidance during this year. I have pestered you countless times about things big and small: the number of times I've come to you with an issue, and you've correctly guessed what it is (and how to solve it!) before I ask is astounding. I've learnt such an incredible amount about formal verification and research as a whole, thanks to both of you.

I also thank you both and Dr. Miki Tanaka for introducing me to formal methods as a field, which has become the thing I look forward to working on at any given moment.

I'd also like to thank my wonderful family, both by blood and otherwise. You may have no idea what I do, or what any of the funny symbols on my computer mean, but you're always just as pumped as I am when things go right, and are always there to give advice and comfort when things don't.

Then, we have all my friends in Trustworthy Systems. Chatting with you and listening to many of you in Monday talks has been a highlight of my week since I first joined. The camaraderie between thesis students in particular was also wonderful to be a part of, sharing in our individual wins and setbacks.

Finally, to single out a colleague, I would like to thank Thomas Liang, my companion through this journey into time protection. Your witty remarks and deep understanding of the project have kept me both sane and able to steadily move forward on my own part of the project. I couldn't have asked for a better counterpart in this endeavour. Fistbump!

Abbreviations

OS Operating Systems

TA Touched Addresses

TCB Thread Control Block

μTC Microarchitectural Timing Channels

L4v L4.verified

ASpec Abstract Specification

AInvs Abstract Invariants

ExecSpec Executable Specification

CSpec Concrete Specification

HOL Higher Order Logic

Isabelle Isabelle/HOL

Contents

Chapter 1

Introduction

Covert channels are a type of security vulnerability that maliciously exploit unintended information leakage to bypass the security policies of a system [Lampson, 1973]. A particular subset of these covert channels, timing channels, bypass the security policies of an operating system (OS) kernel through variations in the timing of observable events, such as the response time of a separate device [Schaefer et al., 1977]. Microarchitectural timing channels (μTCs) are an even more specific type of covert channel relating to kernels [Ge et al., 2018]. They act on shared hardware resources in a system, such as caches, and exploit variations in the timing of operations in these microarchitectural components to violate kernel security policies and enable communication of confidential data.

μTCs have been demonstrated to be practically exploitable, with the Meltdown and Spectre attacks having been reported as recently as 2018 [Kocher et al., 2019, Lipp et al., 2018]. These attacks demonstrated that even secure applications developed according to best practices are capable of being turned into unwitting trojans that leak secrets via μTCs. This highlights the need for kernels to be the main line of defence with regard to security: if the kernel is vulnerable, no application running on top of it can be trusted to be truly secure.

Despite this fact, seL4 is the first kernel to have made a concerted effort towards verifying functional correctness and, importantly for this thesis, verifying the absence of security vulnerabilities [Heiser, 2020]. Unfortunately, whilst other security vulnerabilities, such as storage channels, have been removed from seL4 [Murray et al., 2013], μTCs still have the capability to expose secrets, as no proofs in seL4's proof base, L4v, relate to them.

Ge has proposed a series of OS mechanisms named *time protection* as a method of closing μTCs in seL4 [Ge, 2019]. Amongst other mechanisms, she proposes partitioning caches between separate processes based on the type of cache. Whilst she demonstrated the efficacy of time protection in a modified version of seL4, she found hardware limitations of the x86 and ARM architectures prevented the full closure of μTCs. As time protection has since been implemented in seL4 on top of the RISC-V architecture (with updated hardware) and has subsequently been researched [Buckley et al., 2023], we now seek to formally prove a time protection result in L4v as well.

The aforementioned research by Buckley et al. produced a paper that presents a pathway to achieving this goal. The main suggestion of the proposal relevant to this thesis is to perform *touched address* (TA) *accounting* in all specifications of seL4, where all accesses to memory addresses are tracked as a set of 'touched' addresses. With this TA set, a time protection property can be articulated and proved in L4v.

Separate work by the team at Proofcraft, a commercial group specialising in formal verification, will generate this TA set for the concrete specification of L4v in such a manner that no TA accounting is required in the other specifications. Whilst their work is on multi-core verification, the common nature of the TA accounting required in both projects means their efforts will significantly reduce the work required to verify time protection. Instead of handling this TA accounting, the required task becomes showing that any pointers accessible in a given security domain only point to accessible memory.

This property can be proved by maintaining an invariant from kernel start-up in an abstract specification of seL4. This invariant would enforce that any kernel objects accessible to a security domain only reference other accessible regions of memory. This would directly show that any touched addresses lie within valid security bounds of memory, replacing the prior work to add TA accounting to enforce this property.

This report includes the following contributions to proving a time protection property in L4v:

- A discussion of prior work in formalising spatial partitioning, as a requirement of time protection, in seL4, as well as issues with this prior approach (Chapter 3).
- A discussion of a novel approach to formalising spatial partitioning (utilising the aforementioned invariant condition) and refining this result to the concrete layers of L4v (Chapter 4).
- An account of how this invariant condition was constructed, and proved to be invariant across a sample of L4v's abstract specification, along with a discussion of the process of proving these invariance results (Sections 5.1 and 5.2).
- An account of how this invariant can be related to the intermediate specification in L4v, presenting example proofs of this process and a pattern to simplify this proof process for the remaining sections of the abstract specification (Section 5.3).
- A description of how this work can be extended to eventually prove invariance across the entire abstract specification, and how this result can be refined down to the intermediate specification (Chapter 6)

Chapter 2

Background

Before continuing with an explanation of the goals of this thesis, why it is important and what was achieved, we first discuss some concepts that are integral to understanding the remainder of this report.

Section 2.1 introduces what covert channels are, building up to a definition of a microarchitectural timing channel, and explores why their closure is something to strive for. Section 2.2 introduces formal verification and several adjacent topics utilised in this thesis, and discusses the reasons for their usage.

2.1 Covert Channels

A *covert channel* is an unintended pathway for information flow that is exploited by a malicious actor [Lampson, 1973]. Within an OS, this is often a way for a malicious process to violate security policies and gain sensitive information. Such channels are clearly problematic as they undermine the ability of a system to enforce strict isolation between separate processes, effectively ensuring no sensitive process can be trusted to run on the system.

Covert channels are classifiable based on the mechanism through which they leak data. Physical channels leak data through some physical means, whether that be temperature [Masti et al., 2015, Murdoch, 2006], power consumption [Kocher et al., 1999], electromagnetic radiation [Genkin et al., 2015, Quisquater and Samyde, 2001] or sound [Backes et al., 2010, Genkin et al., 2014].

As an example, a secret key used during RSA encryption can be deciphered from the two different operations used during modular exponentiation: squaring only on a 0 bit, or squaring and multiplying on a 1 bit [Kocher et al., 1999]. As the two different operations consume different amounts of power, an attacker can deduce the bits of the secret key by measuring the power consumption of the machine over time.

Physical channels generally leak information from a system to an external agent who requires physical access to the machine, making it a concern of physical security (which is beyond the scope of a kernel's responsibility).

On the other hand, there exist covert channels which leak information **within a system**, enabling an attacker process to gain access to confidential secrets. These internal channels directly concern the kernel, as it manages the execution of separate processes and is responsible for enforcing isolation between them. These kinds of channels can be broken down into two further subcategories: *timing channels*, and *storage channels* [Schaefer et al., 1977].

Timing channels work by analysing the timing of certain key events in order to glean information. An example of a timing channel is presented in Snippet 2.1.

Snippet 2.1: Password Checker with Timing Channel

```
char *correct_pw = "abc";

int check_pw(char *input) {
    for (int i = 0; i < strlen(input); i++) {
        if (i >= strlen(correct_pw)) return 0;
        if (input[i] != correct_pw[i]) return 0;
    }
    return 1;
}
```

Based on the length of the correct prefix of the guessed password, the password checking algorithm has a variable runtime. Variations in the timing of the algorithm can therefore leak information about the password, with longer runtimes meaning a longer prefix of the guess is correct.

On the other hand, storage channels transfer information through the modification of a storage location in a manner that violates security policy, enabling the transfer of sensitive data between processes. For example, a privileged user might modify the metadata of a file in such a way that an unprivileged user might be able to glean confidential information from it.

Whilst both categories of covert channels are a security vulnerability that should be addressed, timing channels in particular have also been demonstrated to be exploitable remotely, making the endeavour to remove them from seL4 even more critical [Brumley and Boneh, 2003].

2.1.1 Microarchitectural Timing Channels

The specific kind of timing channel we discuss in this report is the *microarchitectural timing channel* (μTC) [Ge et al., 2018]. This is a class of timing channels which specifically targets hardware state in order to facilitate uncontrolled information flow between processes. Such channels utilise the shared nature of microarchitectural state such as caches, which are often abstracted away from the OS itself.

μTCs are distinct from algorithmic timing channels such as the one facilitated by Snippet 2.1. An OS is unable to control the processes running on top of it, and so those forms of timing channels cannot be avoided at the OS level, while μTCs can be more directly addressed by the OS and kernel.

μTCs often arise from well-intentioned efforts to improve the average case performance of systems by optimising for temporal and spatial locality using caches. Since these caches are shared across all processes and are also directly responsible for timing variations based on whether a memory access *hits* or *misses* the cache (i.e. whether the address is in the cache), there is very clear potential for these timing variations to leak secrets, which are μTCs.

The Spectre and Meltdown attacks utilised μTCs in order to break isolation between processes, as well as break isolation between processes and the OS [Lipp et al., 2018, Kocher et al., 2019]. Other attacks make use of known techniques such as *prime and probe* and *flush and reload* in order to achieve their exploits [Osvik et al., 2006, Yarom and Falkner, 2014]. These attacks demonstrate the need for mitigation methods aimed at closing μTCs at the kernel level.

2.2 Formal Verification

Formal verification is the process of mathematically proving results about a given system. These results are described in a *logic*, a methodology of reasoning about the veracity of statements. Examples of logics include *Higher Order Logic* (HOL) or *Zermelo–Fraenkel set theory* (ZF).

Formal verification is preferred over other methods of reasoning about the correctness of programs and systems (e.g. testing) for its strong guarantee of correctness, for some definition of 'correct' determined by a *specification* of intended behaviour. As all reasoning is done mathematically, there is no potential for a program to exhibit unexpected or undefined behaviour beyond what the specification of the program dictates.

Note that this does **not** prevent the specification from being incorrect, which is a common source of bugs within formally verified programs. As an example, when formally verifying a sorting algorithm, its specification may specify that the output of the program is sorted. The program may contain logic errors even with this specification, as it might return no elements, which trivially satisfies the condition even though it is not the intended behaviour.

2.2.1 Hoare Logic and Invariants

Hoare Logic

Within this thesis, we will often discuss *invariants*, a concept stemming from Hoare logic. Hoare logic is a formal system which specifically reasons about programs in terms of *preconditions* and *postconditions* [Hoare, 1969]. Preconditions are requirements of the state of the program *before*

a function call. If the preconditions are satisfied, then we get guarantees about the state of the program *after* the function call, called postconditions.

The syntax for this logic is $\{|P|\}\ f\ \{|Q|\}$ with P being the preconditions, Q being the postconditions and f being the function call in the system. These three components are collectively labelled as a *Hoare triple*. As an example, the Hoare triple

$$\{|\text{True}|\}\ x := 6\ \{|x \neq 7|\}$$

says that for any initial state (since the precondition True is always satisfied), setting x equal to 6 gives a guarantee immediately afterwards that x is not equal to 7.

Invariants

We can extend this logic to introduce the concepts of invariants. We consider an invariant to be any result about the state of the program that **remains true throughout execution**. That is, a predicate P about the state of a program is an invariant if it can be shown true for an initial state of the system and if, for all functions f in the system, $\{|P|\}\ f\ \{|P|\}$ i.e. the property is maintained across all function calls once it is true.

Weakest Precondition

There are multiple ways of transforming proofs about Hoare triples into more conventional implication proofs, with one of the more common methodologies being the *weakest preconditions* transformer. It works by reasoning about the weakest possible precondition that could generate a given postcondition in a Hoare triple [Dijkstra, 1975].

For example, if we have a function f and a postcondition Q, the weakest precondition is the predicate P such that

$$\{|P'|\}\ f\ \{|R|\} \iff P' \implies P$$

That is, P' is a valid precondition for the Hoare triple if and only if P' implies the weakest precondition P.

To generate a weakest precondition for a given function and postcondition, we treat the function in question as a sequence of instructions. We can progressively work backwards from the final instruction to the first, taking the postcondition at each step and generating the weakest precondition for that instruction. We then take that precondition and treat it as the postcondition of the previous instruction to generate a new weakest precondition, repeating this process and working backwards until we have the weakest precondition for the very first instruction. This generated precondition is then the weakest precondition for the pair of that entire function and that particular postcondition.

With this generated weakest precondition P, we know that any possible precondition for the Hoare triple must imply P. As such, we can perform a more standard implication proof using

traditional proof techniques to show that the desired precondition P' implies the generated weakest precondition P.

2.2.2 Refinement and Correspondence

When coding a program to complete a task, the end goal is to have an executable binary that performs that task. However, directly coding this binary, whether as binary or even as assembly, is avoided in many contexts as it is hard to perform higher level reasoning about the program. Programming languages provide abstractions that enable a programmer to understand their code at a higher level without needing to reason about the low level implementations of each command or function. This enables the programmer to reason about and write more complex programs.

Similarly, when formally verifying a program, the final binary must be verified to match a specification, but performing proofs directly on the binary is difficult as the low level implementation details are generally hindrances to verification and formal reasoning. Instead, one can prove results on a higher level implementation of the program which abstracts over the hardware implementation of certain methods and is therefore easier to reason about. This simplifies the process of proving that the program adheres to a given specification.

However, this abstraction serves no purpose unless we can relate it back to the binary. Just as a programming language can be compiled (or interpreted) into machine code, an abstract specification can be *refined* into a concrete (binary) implementation (e.g. [de Roever and Engelhardt, 1998]). In this manner, the results that come about at the abstract level can be reutilised in the concrete layer without the difficulty of directly verifying the binary.

Correspondence

One way to perform this refinement, which is utilised by seL4's refinement proofs, is through 'correspondence' proofs relating an abstract and concrete specification [Cock et al., 2008, Klein et al., 2009, Klein et al., 2010]. A correspondence proof shows that if the two programs start in two 'related' or equivalent states (where each pair of corresponding fields in the states are effectively equivalent), then the possible final states after the function calls are also related. It achieves this by analysing each step of the corresponding function calls and proving that the state relation is preserved at each step, with the correspondence proof for lower level methods being utilised to show correspondence for higher level methods.

These correspondence proofs can then be 'chained together' to show that any two sequences of corresponding function calls maintain this state relatedness. By doing this, it can be proved that the two specifications correspond for the entire runtime of the programs, regardless of which functions are called and in which order. Figure 2.1 demonstrates this process: the initial states $\mathtt{s}_1$ and $\mathtt{s}'_1$ are related, and the resultant states after the function call $\mathtt{f}_1$ and $\mathtt{f}'_1$ remain related, and so on.

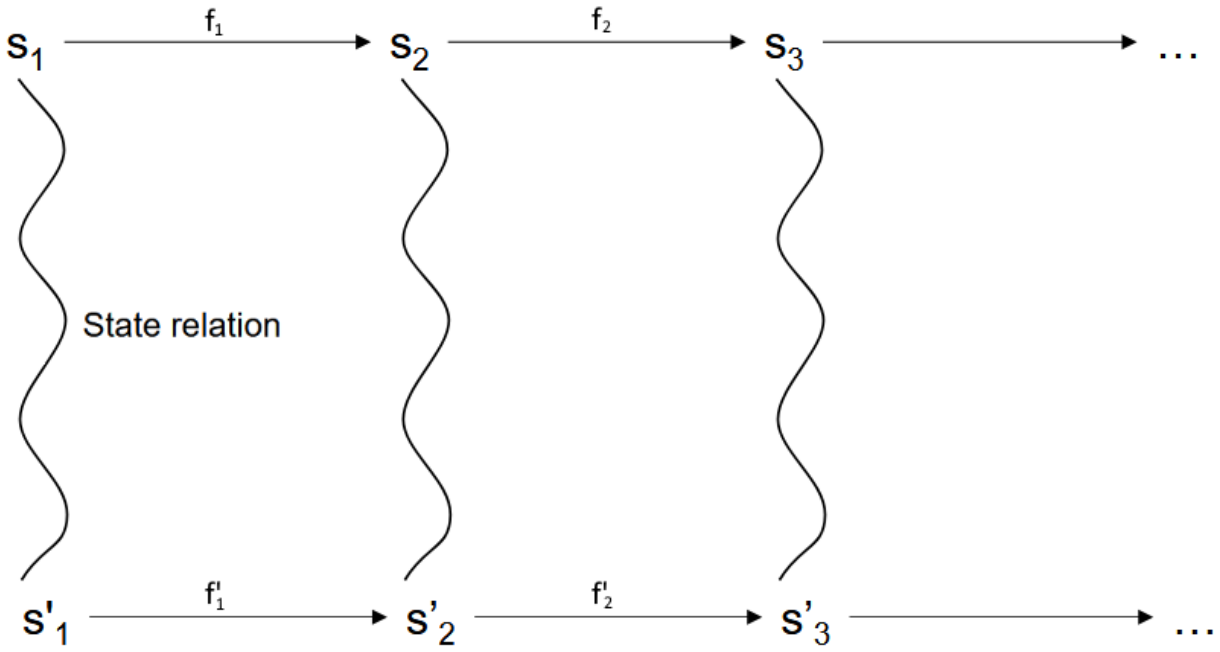

Figure 2.1: Chaining Correspondence Proofs to Relate Abstract and Concrete Executions

In this manner, we can show that the two specifications are proper counterparts and successfully refine the abstract specification into the concrete implementation. In particular, this enables us to show that the execution of the abstract program *simulates* all possible concrete program executions paths, i.e. for each concrete program execution path, there is a corresponding abstract program execution path. This means that any results proved about the abstract specification also apply to the concrete implementation, 'refining' the results as desired.

Chapter 3

Prior Work

With the discussion of fundamental topics complete, we now discuss the foundational work which this thesis relies upon: seL4 (Section 3.1), Isabelle/HOL (Section 3.2) and L4v (Section 3.3).

Regarding the particular topic of verifiably closing μTCs at the kernel level, the efforts of Ge and Buckley et al. are currently one of a kind to the best of my knowledge [Ge, 2019, Buckley et al., 2023]. Whilst there are other projects which seek to reason about timing related information flows [Arm Limited, 2024, Braun et al., 2015, Brickell et al., 2006, Bernstein, 2005, Liu et al., 2016], there are none relating to specifically kernel level verification of the closure of μTCs.

As a result, we will discuss the prior work on closing μTCs on seL4 in particular, and discuss potential drawbacks of the approach proposed by Buckley et al. (Section 3.4).

3.1 seL4

seL4 is a member of the L4 microkernel family being developed by Trustworthy Systems, a research group specialising in systems and formal methods research at the University of New South Wales [Heiser and Elphinstone, 2016]. It is 'a high-assurance, high-performance operating system microkernel' which aims to provide security guarantees without sacrificing performance, boasting two to ten times better results on key benchmarks compared to other microkernels [Heiser, 2020].

From a formal verification standpoint, seL4 uses interactive theorem proving with a proof assistant in order to verifiably guarantee results about the microkernel. These results range from functional proofs about correctness of operations, to security proofs, which show that the kernel is able to sufficiently isolate separate processes running on top of it. With these proofs, we can trust that, short of hardware failure, seL4 will always perform to those guarantees.

To reason about information flow, seL4 implements *security domains*, which are a construct used to isolate independent subsystems [Trustworthy Systems Team, 2017]. Security domains

are used to control and limit the information flow between these independent subsystems. A thread will belong to exactly one domain, and only runs when the domain is active. This is a useful concept to note when discussing information flows between separate processes, as will be necessary when discussing the closure of μTCs.

3.2 Isabelle/HOL

Proofs about seL4 make use of an interactive theorem prover named *Isabelle* in order to facilitate the formal verification of seL4. Isabelle is a generic theorem proving system, meaning it provides a metalogic with which logics can be implemented [Nipkow et al., 2002].

Instances of Isabelle with these logics implemented (e.g. Isabelle/HOL, Isabelle/ZF [Paulson, 2013]) can then use the implemented logic to write formal logical statements and reason about them. These statements can be used to formally verify results about programs or mathematical concepts.

Isabelle is also interactive, as the user guides the proof process with commands to modify the proof state, and Isabelle programmatically checks that the sequence of user provided instructions constitutes a valid proof of the stated property. This check provides a guarantee of the correctness of the result.

These mechanically verified proofs rely on a small 'trusted proof base' upon which further proofs are built upon, hence providing guarantees about a formally verified program that require very little trust. This is a critical requirement for seL4 which aims to make guarantees about the kernel's behaviour so that, true to Trustworthy Systems' name, the kernel is worthy of trust.

3.3 L4v

L4.verified (L4v) is a project built on top of seL4, which aims to formally verify the microkernel's correctness and security using Isabelle/HOL (henceforth referred to as Isabelle) [Klein et al., 2009, Klein et al., 2014]. L4v makes extensive use of Isabelle *sessions* and *locales* (logical groupings of theories, proofs and assumptions) to methodically and comprehensively provide guarantees about seL4 in a way that is modular (with locales depending on and referring to other locales) and better organised than one prohibitively large proof environment.

3.3.1 Models, Specifications and Correspondence

L4v contains several layers of models of seL4, each with their own specification of expected behaviour, and correspondence proofs between these specifications.

Figure 3.1 shows the three specifications: the *abstract specification* (`ASpec`), the *executable/design specification* (`ExecSpec`) and the *concrete specification* (`CSpec`). The `ASpec` describes the

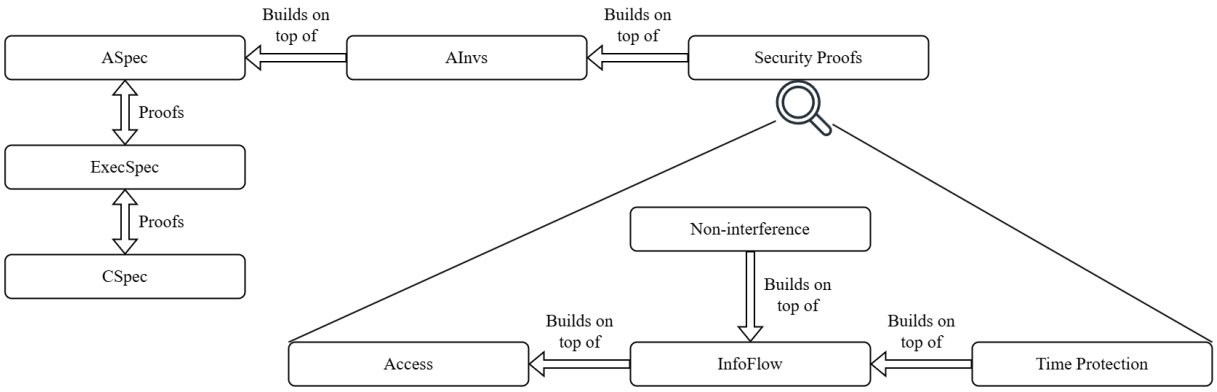

Figure 3.1: Structure of specifications and related proofs

behaviours expected of the kernel in Isabelle, with several other sessions built upon it to prove results about this abstract representation. The `ExecSpec` acts as an intermediate representation of the kernel, describing an implementation of the kernel in Haskell. Finally, the `CSpec` reasons about a version of the C code that implements seL4: the C code is parsed and imported into Isabelle, and is reasoned about in Isabelle. With these three specifications and correspondence proofs between them, we can reason about the C implementation of seL4 at different layers of abstraction.

Figure 3.1 also shows the extra Isabelle sessions built on top of the `ASpec` in particular. The *abstract invariants* (`invs` in the Isabelle session `AInvs`) are a collection of invariants which are maintained throughout the execution of the `ASpec`, and which can be used to prove results in the `ExecSpec` and `CSpec` during refinement, which is discussed in Section 3.3.2.

The security proofs are the other set of sessions built on top of the `ASpec` and `AInvs`. They verify results related to security policy management, enforcing the confidentiality and integrity of privileged information. In particular, *InfoFlow* is a set of proofs controlling the available channels of information transfer between two security domains, and between the OS and a given process. This session proves the absence of covert channels such as storage channels, and is where proofs regarding the closure of μTCs should be built upon.

3.3.2 Refinement

L4v makes use of refinement to relate its `ASpec`, `ExecSpec` and `CSpec` specifications without repeating the same proofs at each layer [Klein et al., 2010]. Figure 3.1 visualises the correspondence proofs that perform this refinement, which are completed in two Isabelle sessions named `Refine` (between the `ASpec` and `ExecSpec`) and `CRefine` (between the `ExecSpec` and `CSpec`).

For this thesis, we focus on the `corres` statements in `Refine` which relate the `ASpec` and `ExecSpec`, written as follows.

Snippet 3.1: Shape of `corres` Lemmas

```
proposition corres_lemma:
    "corres ret_rel
        P P'
        e e'"
```

In this statement, `e` and `e'` are the two expressions which are stated to be equivalent in the two specifications. `P` and `P'` are the preconditions under which the expressions are equivalent. As an example, `get_tcb` is only valid to use if there is a thread control block (TCB) at the address provided, so that is a necessary precondition for a `corres` proof relating to it. Finally, `ret_rel` is a method that relates the values returned by the two expressions, checking whether they are equivalent in the two specifications. For this thesis, the expressions and preconditions are the main parameters of note.

Assertions

A topic of particular note for this thesis is the fact that assertions can be added at each layer, asserting that a certain condition must be true when the assertion is reached during execution. An assertion provides guarantees at the layer in which it is added, but its condition must also be proved true at some stage, as is typical in formal verification.

This proof obligation gets pushed to the correspondence proofs, going to `Refine` for assertions in the `ExecSpec` and `CRefine` for assertions in the `CSpec`. With correspondence proofs, we can instead show the asserted condition is true in the higher level specification, and that it corresponds to the assertion being true in the lower level specification, successfully refining this property down. This presents a way for the proof obligation to get pushed up a layer to a higher level specification.

In this thesis, we focus on Thomas Liang's addition of assertions to the `ExecSpec` layer, which he did using a method named `stateAssert` [Liang, 2025].

3.4 Time Protection

All of this explanation leads to the goal at hand: verifiably closing μTCs in seL4. Ge has proposed a methodology called *time protection* to achieve this goal, and has implemented it on an experimental version of seL4 [Ge, 2019], both on top of x86 and ARM architectures. Her work serves as a good proof of concept for time protection, demonstrating its potential as a method of closing all μTCs in seL4.

Time protection has several requirements on how the kernel must operate in order for it to ensure the absence of μTCs. Most of these are listed without further explanation as they do not impact the work of this thesis, and are already well discussed and explained in Ge's PhD thesis:

1. When time-sharing a core, a context switch between domains must force any microarchitectural state to be reset to a predefined 'reset' state unless the state supports spatial partitioning.
2. Each domain must have a separate copy of the OS text, stack and (as much as possible) global data.
3. Access to other shared OS data must be made deterministic.
4. State flushing must be padded to its worst-case latency.
5. When sharing a core, the OS must disable or partition interrupts other than the preemption timer.

The main concern for this thesis is the first requirement, part of which requires that for some *spatially partitionable* microarchitectural state, such as off-core caches, the resource is spatially partitioned between security domains by *cache colours*. Cache colours are a partitioning of physical pages in memory, with colours uniquely defining where in the cache a page can lie.

Hence, part of the first time protection requirement is that the system must enforce that, for spatially partitionable state, separate security domains must occupy separate regions of the cache, with no overlap between these regions. This requires that each security domain is only allowed to touch the regions of memory associated with its allocated cache colour(s), with no colour allocated to two security domains.

The second requirement presents a motivation for why this spatial partitioning is necessary. Enforcing that each domain only accesses its own copy of the OS text, stack and global data requires spatial partitioning of the copies of this state. This in turn motivates enforcing that each domain's copy of this state is written to its allocated cache colours to ensure spatial partitioning.

The rest of the first requirement requires that for any other state which **cannot** be spatially partitioned, it must be *temporally partitioned*, meaning that on a context switch from one security domain to another, the shared resource must be reset to some predefined state. These two parts of the first requirement ensure that the only state accessible to a security domain is the state which it has been allocated, preventing two such domains from sharing state (with only a few necessary exceptions).

This thesis takes a focus on proving the spatial partitioning of memory through the allocation of cache colours to security domains.

3.4.1 Touched Addresses

Buckley et al. have suggested a methodology for implementing a proof of this spatial partitioning of memory (amongst other suggestions for proofs of all the requirements of time protection) in a follow-up paper to Ge's original time protection paper [Buckley et al., 2023]. In their paper, they suggest performing *touched address accounting*. This involves keeping track of any addresses that are read from or written to, which we consider *touched addresses* (TAs). We keep track of these

TAs at each of the three specifications in L4v, and then prove subset relations between the TA sets, namely that

$$\mathrm{TA}_{\mathrm{CSpec}} \subseteq \mathrm{TA}_{\mathrm{ExecSpec}} \subseteq \mathrm{TA}_{\mathrm{ASpec}}$$

This enables the `ASpec` to reason about the potentially touched addresses using a safe overapproximation of the actual touched addresses. This enables a proof showing that the touched addresses all lie within the set of addresses accessible to a given security domain (determined by cache colours). This can be articulated in Isabelle by showing the overapproximation (as a set) is a subset of the set of all addresses accessible to the domain. Buckley et al. began working to complete this proof in 2023 within a time protection Isabelle session extending off of the existing InfoFlow proofs, as seen in Figure 3.1.

For this discussion of background, we restrict the scope of discussion to TA set accounting at the `ASpec` layer, as that is the layer most directly relevant to this thesis. TA accounting at the `ASpec` entails adding a field to the program state to track the predicted TA set, and modifying all functions to update the TA set whenever an address is potentially touched.

This 'prediction'/overapproximation is necessary as the `ASpec` is abstract and should remain agnostic of implementation details. Hence, it will not be able to follow an exact execution path to decide which addresses are touched. As such, Buckley et al. suggest maintaining an overapproximation as it allows the result to be proved on *all* traces which touch addresses lying within the overapproximation. This means that implementing assertions that touched addresses lie within this overapproximation will suffice, instead of having to reason about exact traces.

3.4.2 Issues with Prior Work

During the implementation of this TA accounting in 2023, several issues with the proposal arose. In the `ASpec`, around 125 methods had to be updated extensively, adding assertions to correctly track potentially accessed addresses [Buckley and Sison, 2023].

More critically, in the process of implementing this TA accounting, well over 900 proof breakages were observed in `AInvs` and the security proofs. These breakages were seen in the form of lemmas that needed to be `sorry`-ed, meaning that the proof engineer effectively asserted that the lemma was true without proof.

The reason for these breakages was that certain invariants relied on the fact that some methods do not update the state of the model during execution. That is, they assumed that for certain functions f,

$$\{|P|\}\ f\ \{|P|\}$$

for all predicates P. After parametrising the TA set into the `ASpec`, however, these invariants no longer held, as these functions would access memory addresses. This resulted in the TA set being updated, which constituted an update to the state.

Fixing this issue required rephrasing these properties to utilise the fact that the relevant functions did not modify the state *modulo TA accounting* i.e. not including changes to TA accounting.

This was an arduous task, and required an estimated one to two years of development, with at least 9 months' worth of work remaining by the time the project was put on hold.

Chapter 4

Methodology

We now discuss an updated approach to verifying spatial partitioning in L4v, motivated by a novel proposal detailed in Section 4.1. This proposal leads to a modified proof process, changing what the proofs at each specification layer look like, which is discussed in Section 4.2.

4.1 New Development

As discussed in Section 3.4, the prior plan of action had a large time commitment dedicated to simply fixing the breakages created during the updates to the `ASpec`. A new proposal from Proofcraft, briefly discussed in Chapter 1, removes the need for handling this accounting at the `ASpec` level. Their separate work on multi-core support for seL4 similarly requires TA accounting, but their work renders TA accounting at the `ASpec` level infeasible, as it would interfere with existing machinery they have developed.

As such, they have decided to perform TA accounting and directly show that this TA set lies within the cache colour allocation, all at the `CSpec` level. Whilst this work is yet to begin, the result of this work would be usable in the formalisation of time protection. This would remove the need for TA accounting in the higher level specifications and save years' worth of proof work in resolving the aforementioned breakages.

Since we now need to ensure that TAs lie within their allocated cache colours in the `CSpec` itself, Thomas Liang, as part of his thesis, moved the relevant assertions into the `CSpec`, where pointer guards already perform checks on every memory access [Liang, 2025]. He extended these pointer guards with a check that any memory access must target an address within the colour allocation of the current security domain, which removed the need for separate assertions in higher level specifications.

However, this guard still requires a proof that the added condition holds.

4.2 Cache Colour Allocation

L4v currently relies on abstractions representing the capabilities of a given security domain in its security proofs, in order to determine which memory can and cannot be accessed by a given process. As the cache colour allocation dictates which regions of memory security domains can access, these security proofs are the logical place to implement this allocation. Since the security proofs are built upon the `ASpec`, any properties relating to this allocation should likewise be written on top of the `ASpec`.

To show that the `CSpec` guards' condition holds, we need to introduce a corresponding property in the `ASpec` that can be used to prove this condition. As the team at Proofcraft is now pushing the proof that the TA set lies within the cache colour allocations down to the `CSpec`, we no longer need to consider a TA set when deciding this property. Instead, we can state a property that directly reasons about *kernel object references*, since these references determine which memory locations are accessed.

Property 1 (Kernel Object Property)**.** *All kernel objects in the allocated colour(s) for a given security domain only reference other addresses within the allocated colour(s) for that domain.*

With this property, we can reason about what addresses can be touched at all, presenting a pathway to showing the TA set lies within the cache colour allocation at the `CSpec` level.

In order to prove this property, we can phrase it as an invariant in the `AInvs`, which is maintained throughout the running of the abstract specification of the kernel. The process of proving this invariant is discussed in Section 4.2.1, and the invariant's definition in Isabelle is presented in Chapter 5.

By ensuring this invariant is always maintained, it can be utilised during refinement to indirectly prove that the conditions added to the pointer guards in the `CSpec` are true, which is discussed further in Section 4.2.2.

4.2.1 Proving the Kernel Object Invariant

Proving that an invariant is maintained requires showing that it is true at some state (either on program start-up or shortly after) and that all methods in the specification maintain the invariant.

The kernel object invariant only satisfies the former requirement **after system initialisation**. During system initialisation, a root process allocates capabilities for **every other process**, violating the kernel object property by modifying other security domains' memory and violating spatial partitioning. This is naturally a concern, as this requires some way to 'switch on' the invariant after system initialisation, instead of verifying it to be true from start-up.

After system initialisation, however, we can make an assumption about the resultant state, asserting that the system initialiser allocates the capabilities for each domain based on the

cache colours allocated to it. There is precedent for making such an assumption, as the existing security proofs in L4v assume that the system initialiser configures the system's capabilities in a manner that enables information flow control to be enforced [Murray et al., 2013]. By similarly assuming that the system initialiser provides a well-formed capability state for us to work with in our time protection work, we satisfy the first requirement to prove the invariant.

To demonstrate that the invariant is continually maintained, we need to prove that it is maintained by the `ASpec` method `call_kernel`. The common pattern of proving such abstract invariants is to complete proofs on lower level methods, and compose those proofs to prove invariance across higher level methods, building up towards `call_kernel`.

The proving of this invariant is also the place that the prior time protection work of Buckley et al. is most likely to be usable, as both the invariant and the TA accounting relate to checking memory accesses. As such, part of the work of proving this invariant is to investigate to what extent the existing infrastructure can be reused.

4.2.2 Refining the Kernel Object Invariant

Once the invariant is proved over the `ASpec`, it then needs to be refined down to the `CSpec`, where the pointer guards are. As discussed in Chapter 3, properties in the `ASpec` are refined down to the `ExecSpec` via `corres` proofs in `Refine`. In particular, assertions in the `ExecSpec` will provide guarantees at the `ExecSpec` layer, whilst requiring a proof of the assertion's validity, which can be discharged via a `corres` proof.

Hence, we can refine the kernel object invariant down to the `ExecSpec` with two steps. First, we add an assertion to the `ExecSpec` that, in a given method, an accessed memory address lies within the cache colour allocation of the current domain. Then, we add a `corres` proof which utilises the invariant to show that the `ASpec` implementation of the method matches this behaviour, also maintaining the assertion condition.

Similarly, to refine the kernel object property down to the `CSpec`, we utilise the concrete counterpart to `corres`, `ccorres` proofs. The assertion in the `ExecSpec` can be refined down to the `CSpec` via `ccorres` proofs in `CRefine`, with the guard/assertion at each of the two layers directly corresponding with each other. This makes for a more obvious correspondence connection (even if the proof itself is not made easier). This `ccorres` proof will satisfy the guard conditions, completing the refinement process.

In this manner, the abstract invariant can be refined down to discharge the concrete guard conditions, successfully proving the time protection requirement that the `CSpec` TA set is a subset of the cache colour allocation.

4.2.3 Oracle

Another requirement to proceed with proving the kernel object invariant in the `ASpec` is that we must have an implementation of the cache colour allocation. At the moment, there is no

clear pathway to determining this allocation as the `ASpec` and its security proofs do not have any reference to the concept of cache colours, since this is one of the implementation details abstracted over.

Without such a method to determine the colours accessible to a given security domain, the rest of the project cannot progress. Hence, we need to construct an *oracle function* that is assumed to perform such a mapping correctly, without providing an implementation. Naturally, the true colour allocation will satisfy certain conditions which we will need to utilise, such as having no overlap between domains, and so we will also need to assume this oracle will also satisfy these properties.

The oracle function can be used in proofs to do this domain-colour mapping, standing in for an implemented function. This ensures that any implementation which satisfies the oracle's requirements will be able to replace the oracle, including the desired allocation when it is developed.

With this colour oracle, progress can be made on the other tasks of proving invariance across the `ASpec` and refining the invariant to the `ExecSpec`, whilst deferring the investigation into how to implement such an allocator. Due to the differences between the `ASpec` and `ExecSpec`, two separate oracles will actually be needed, though in principle they should represent the same underlying allocation. Relating these separate oracles will be discussed in Chapters 5 and 6.

4.3 Aims

With the `ASpec` invariant, the two levels of refinement and the `CSpec` pointer guards, there is a clear point of delineation at the `ExecSpec` for two theses: this thesis, and Thomas Liang's thesis. Thomas Liang was tasked with investigating the `ExecSpec-CSpec` refinement and adding assertions to both layers [Liang, 2025], and so the remaining parts of the task are the aims of my thesis, which are to determine

1. the feasibility of proving an invariant that establishes Definition 1 in the `ASpec`, both in terms of setup and proofs.
2. the feasibility of refining the invariant condition to satisfy the `ExecSpec` assertions added by Thomas Liang, noting any modifications that need to be made to the assertions for refinement to work.
3. whether prior time protection work in TA accounting is reusable.

Chapter 5

Results

We now discuss how these aims were met during the course of this thesis, and the work done to achieve these goals.

In Section 5.1, we discuss the process of articulating the kernel object invariant condition in Isabelle. We then discuss proving the condition's invariance across several representative samples of the `ASpec`, and the proof patterns utilised in doing so, in Section 5.2. Finally, we discuss the refinement of this invariant condition to the `ExecSpec`, along with the insights gained during the proof process, in Section 5.3.

5.1 Setup for Invariant

5.1.1 Oracle

As mentioned in Chapter 4, in order to articulate the invariant condition, I needed an oracle to determine which regions of memory are allocated to which security domains. This is because there is no clear pathway to providing an implementation for such an allocation in the `ASpec` at this time.

This oracle acts as a stand in for a defined implementation, specifying certain properties that it must satisfy. It can be replaced by an implementation at a later date and as long as the properties asserted of the oracle are shown to hold for that implementation, any proofs relying on this oracle should also work with the implementation.

Snippet 5.1: Colour Allocation Oracle

```
axiomatization
  colour_oracle :: "domain ⇒ obj_ref set"
where
    colour_oracle_no_overlap:
        "x ≠ y ⟹ (colour_oracle x ∩ colour_oracle y = {0})"
```

Snippet 5.1 shows how the oracle is *axiomatised* into the Isabelle theory file. This means it is effectively taken for fact that such a method exists which satisfies the properties underneath. Axiomatisation is risky to use as the axiomatised method is not checked at all and hence presents a way to introduce contradictions into the trusted proof base (e.g. if two contradictory properties are declared for the oracle).

Axiomatisation is necessary in this case as it presents a way of creating an oracle that is simply asserted to exist without needing to provide a definition, instead only providing properties. So far, the only property added to the colour oracle is that no two security domains have overlapping regions of memory assigned to them (excluding 0, or null). As further properties are identified which the oracle must satisfy, they can be added to the list of its properties and will then be imposed on the replacement implementation.

5.1.2 Helper Methods

With the oracle defined, I could use it to define methods to help with articulating the invariant condition. These methods should check that all references that a kernel object makes to other addresses should lie within the cache colour allocation of the current security domain.

Creating these definitions ended up being a fairly mechanical process. It involved iterating through each type of kernel object, identifying what object references it has as part of its state, and adding a condition to the definition checking that all such references lie within the set of valid addresses.

A snippet of the kernel object check's definition is presented in Snippet 5.2 to provide examples of what the definition looks like and hint at how the definition was generated. The remaining cases are elided as they do not provide further insights. The capability check is defined in terms of an existing method `obj_refs`, which performs a similar accounting of the references contained by a capability, and returns the set of these references.

Snippet 5.2: Kernel Object Helper Methods

```
primrec check_kernel_object_ref :: "kernel_object ⇒ obj_ref set ⇒ bool"
    where
  "check_kernel_object_ref (Endpoint ep) obj_dom = (
      case ep of
              IdleEP ⇒ True
            | SendEP s ⇒ (set s ⊆ obj_dom)
            | RecvEP r ⇒ (set r ⊆ obj_dom)
   )"
 | "check_kernel_object_ref (CNode _ cs) obj_dom = (
    ∀x. case (cs x) of
          Some cap ⇒ check_cap_ref cap obj_dom
        | None ⇒ True
    )"
 ...

primrec check_cap_ref :: "cap ⇒ obj_ref set ⇒ bool"
    where
"check_cap_ref cap obj_set ≡ obj_refs cap ⊆ obj_set"
```

The definition was not defined directly in terms of the oracle. Instead, it accepts the set of valid addresses as a parameter. This was done in case there ended up being other potential uses for the method, which relied on sets of addresses which did not come from the colour allocation. In the end, however, this situation was not encountered.

5.1.3 Articulating the Invariant

With the helper methods defined, the invariant condition was ready to be articulated. Over the course of this thesis, the definition underwent several iterations, with each iteration being developed based on my understanding at the time. Every progression to a new definition was spurred by issues that arose with the prior definition.

Initially, the condition only specified one pointer and kernel object as parameters, as seen in Snippet 5.3. This meant that any methods which handled several kernel objects needed to specify each one as a separate clause in its preconditions and postconditions.

Snippet 5.3: Initial Colour Invariant Articulation

```
definition colour_invariant
  where
    "colour_invariant ptr kobj s ≡
    (ko_at kobj ptr s ∧ ptr ∈ colour_oracle (cur_domain s)) ⟹
    check_kernel_object_ref kobj (colour_oracle (cur_domain s))"
```

This invariant condition had two issues. First, this required manually deciding which pointers and kernel objects needed clauses, which was tedious and error prone. Second, as the condition would differ for each method based on the references the method touches, proving this version of the property was invariant would be difficult, if not impossible.

Hence, I generalised the definition to require the condition to be true for all kernel objects and their pointers, generating a property that I expected I could prove was invariant across the `ASpec`.

Snippet 5.4: Second Colour Invariant Articulation

```
definition colour_invariant
  where
    "colour_invariant s ≡ ∀ ptr kobj.
    (ko_at kobj ptr s ∧ ptr ∈ colour_oracle (cur_domain s)) ⟹
    check_kernel_object_ref kobj (colour_oracle (cur_domain s))"
```

Snippet 5.4's definition of the invariant condition served its purpose well for a majority of the thesis, facilitating demonstrative proofs which showcased the proof process, as well as how it could be automated (discussed in Section 5.2.2).

However, when it came time to attempt proving invariance across `schedule`, the `ASpec` method which determines which security domain should run next, it became evident that this invariant condition was not general enough. It only stated that the kernel object property was true *for the current domain*, which meant that when it came time to switch domains, the new domain may not satisfy the condition. Hence, `schedule` would not maintain the invariant without extra side conditions.

Therefore, the invariant was generalised one final time to arrive at the current articulation of the invariant.

Snippet 5.5: Current Colour Invariant Articulation

```
definition colour_invariant
  where
    "colour_invariant s ≡ ∀ ptr kobj. ∀ dom ∈ (domain_list s)
    (ko_at kobj ptr s ∧ ptr ∈ colour_oracle dom) ⟹
    check_kernel_object_ref kobj (colour_oracle dom)"
```

Snippet 5.5's definition of the invariant condition states that the kernel object property is true for all domains in the kernel, ensuring that no matter what domain is scheduled next, it will **also** satisfy the property. This definition was implemented relatively late in the thesis, and led to breakages for approximately one third of all my prior proofs. The repair of these breakages will be discussed in Chapter 6.

To the best of my knowledge, this invariant cannot be made more generic as the pointer, kernel object and domain are all universally quantified. No other parameters in the invariant **can** be

generalised, so this should not need to be modified any further.

5.2 Proving Lemmas for Invariant

With the kernel object invariant condition defined, I wanted to prove that it is maintained across `call_kernel`. However, this is too large of a method to tackle at once, and so I first proved maintenance across lower level methods in Section 5.2.1. In doing so, I also utilised some automation machinery discussed in Section 5.2.2.

I then composed those proofs with each other to build up to higher level methods, with the goal of building up towards `call_kernel`, in Section 5.2.3.

5.2.1 Kernel Heap Methods

The lowest level methods present in the `ASpec` are `get_object` and `set_object`, which retrieve and update the referenced kernel object, respectively. Any method which utilises the kernel heap must invoke one of these methods. This makes these methods, along with other methods relating to the kernel heap, the 'building blocks' with which other methods are built.

As a result, I decided to begin my proof work by proving invariance across these lower level methods, as they were easier to reason about and also presented a solid foundation upon which proofs for higher level methods could be built upon. The Hoare triples for these methods all only required weakest precondition style proofs, with any separate complexities coming from variations in the definition.

Notably, most of the 'getter' methods did not require explicit proofs. As they extract information from the state without modifying it in any manner, there were already existing results that **any predicate** would be preserved by a function call. These results trivially prove kernel object invariance for the getter methods, and so no separate proof was required.

On the other hand, some methods required *side conditions*, extra preconditions which had to be included so that the invariant condition is a valid postcondition. For example, `set_object` requires that the object being set also only references addresses in the memory allocation of the current domain, as can be seen in Snippet 5.6.

Snippet 5.6: `set_object` Invariance Lemma

```
lemma set_object_colour_maintained:
    "{|
        colour_invariant and
        (valid_ptr_in_cur_domain ptr) and
        (λs. check_kernel_object_ref kobj (colour_oracle (cur_domain s)))
    |}
    set_object ptr kobj
    {|λ_. colour_invariant|}"
```

In this manner, all the kernel heap methods were either ignored (due to already being directly provable) or were proved with or without side conditions.

5.2.2 `crunch` and Automation

During the kernel heap proofs, a clear pattern became evident. Proofs of invariance over higher level methods would break down into the lower level methods they are composed of, and would use the invariance proofs of **those** methods to build up the proof for the higher level method. Eventually, this pattern would break down when side conditions were introduced or the method was too low level to be broken down (e.g. `set_object`). In these cases, the proof would revert to the aforementioned weakest precondition proof pattern established in Section 5.2.1.

In the cases where there were no side conditions required for the method, this pattern presented an opportunity for automation. This automation came in the form of `crunch`, a piece of machinery developed as far back as 2008 for L4v [Cock et al., 2008]. It had been developed for the purpose of automating this process for any invariant: recursively breaking down methods, proving the invariant across the lower level methods and building a proof back up for the original method. Snippet 5.7 showcases how `crunch` is utilised to prove several methods at once, and how passing lemmas to aid the crunching process works.

Snippet 5.7: `crunch`-ing Several Methods

```
crunch reschedule_required,
       possible_switch_to,
       set_thread_state_act,
       set_priority,
       set_scheduler_action,
       tcb_sched_action,
       set_tcb_queue,
       set_irq_state
for colour_maintained: "colour_invariant"
    (simp: colour_invariant_def obj_at_update)
```

With `crunch`, I was able to simplify the proofs of 14 methods into one line expressions, removing

a couple of weeks' worth of required work. There is also a reasonable chance that other methods which were proved prior to my use of `crunch` can also be automated away. Overall, `crunch` presents a way to reduce the manual workload required in proving the kernel object invariant across the `ASpec`.

5.2.3 Further progress

With the kernel heap methods complete, I decided to approach the problem of proving invariance across `call_kernel` head on. With `crunch`, I was able to attempt to prove `call_kernel` directly. Wherever this proof failed, I would take the method which it failed at, and create an invariance proof for it as a separate lemma. Upon completion of these proofs, the `call_kernel crunch` would then fail in new places, which could be tackled separately again.

Repeating this process presented a quick way to find methods where proofs were needed, instead of needing to scour the specification for methods and see if they would need proofs (or if existing proofs already covered them).

In this manner, over 25 more proofs were completed besides the kernel heap methods, making a good start on building up proofs towards invariance over `call_kernel`. Six of these proofs were completed using `crunch`. The remaining methods were more complex than the kernel heap methods, and had more variation in how they were proved. For example, `transfer_caps_loop` has a recursive definition, which meant the invariance proof for it involved induction. Even in this complex case, however, the underlying proof for the inductive step was a weakest precondition style proof.

Two methods not found via `crunch`, `schedule` and `activate_thread`, also had invariance proofs completed for them. These are two of the three methods which are called directly by `call_kernel`, and were relatively reasonable to prove invariance over. These served as a nice demonstration of the feasibility of proving this invariant for higher level methods. Their proof statements are provided in Snippets 5.8 and 5.9

Snippet 5.8: `schedule` Invariance Theorem

```
theorem schedule_colour_maintained:
    "{|colour_invariant|}
      schedule
    {|λ_. colour_invariant|}"
```

Snippet 5.9: `activate_thread` Invariance Theorem

```
theorem activate_thread_colour_maintained:
    "{|
        colour_invariant and
        (λs. valid_ptr_in_cur_domain (cur_thread s) s) and
        (λs. tcb_at (cur_thread s) s)
    |}
    activate_thread
    {|λ_. colour_invariant|}"
```

Note that `schedule` was the cause of the final change to the invariant, which led to breakages of several proofs. These breakages entailed certain methods (such as `activate_thread` in Snippet 5.9) needing extra side conditions stating that the address being modified is in the current domain (excluding null).

This is always true, but is only proved to be true in the security proofs, which are not accessible in the `AInvs` session. This extra requirement is necessary here, however, because if a method modifies an object at an address in another domain's allocated memory, it will invalidate the invariant condition for that domain, as the modified object needs to point to memory allocated to the current domain.

To add this requirement succinctly to the methods that required them, it was made into a separate helper method stating that a pointer is valid, meaning that it is not null and is in the current domain's colour allocation (as determined by the colour allocation oracle). This method's type signature can be seen in Snippet 5.10. With this method's addition to the preconditions of many proofs, most of the breakages were successfully resolved.

Snippet 5.10: Valid Pointer Helper Method

```
definition valid_ptr_in_cur_domain ::
    "64 word ⇒ 'a abstract_state_scheme ⇒ bool"
```

5.3 Refining to `ExecSpec`

All the proofs at the `ASpec` level are useless if they are unable to discharge the proof obligations created by the assertions added by Thomas Liang at the `ExecSpec` level [Liang, 2025]. As there are existing proofs of correspondence already present in L4v, I aimed to reuse as much of the existing proof work as possible, discussed in Section 5.3.1.

I also tried to identify a proof pattern that would be simple to use and reusable for as many proofs in `Refine` as possible, discussed in Section 5.3.2. I then proved modified correspondence lemmas across a few methods as proofs of concept to demonstrate the proof pattern, as seen in Section 5.3.3

5.3.1 Modified `corres` Statement

To use the abstract invariant to show that the `ExecSpec` assertion condition holds, I needed to modify the `ASpec` preconditions to include the condition that the address is in the correct colour allocation for the current domain. I then utilised this precondition to show that the assertion on the `ExecSpec` side is satisfied.

Proving this modified version of the lemma required an update to the proof itself, adding a proof step to directly demonstrate that the assertion's condition is upheld. This proof step showed that the assertion does nothing during execution, which meant that it could be ignored for the rest of the correspondence proof. This allowed the rest of the original proof to proceed unhindered, reducing the novel proof work that I had to do.

So overall, the correspondence proof for any functions f and f' (in the `ASpec` and `ExecSpec`, respectively) was transformed to include an extra `ASpec` precondition and `ExecSpec` assertion. The shape of the `corres` proofs before and after modification are presented in Snippets 5.11 and 5.12, respectively, with `f''= (stateAssert (λs. isInDomainColour (ksCurDomain s) ptr) [] >> f' ptr)`, a version of `f'` with the relevant assertion(s) added.

Snippet 5.11: Shape of Initial `corres` Lemma

```
proposition corres_f:
    "corres r
        P P'
        (f ptr) (f' ptr)"
```

Snippet 5.12: Shape of Modified `corres` Lemma

```
proposition corres_f_inCurDomain:
    "corres r
        (P and (λs. ptr ∈ colour_oracle (cur_domain s))) P'
        (f ptr) (f'' ptr)"
```

With this modified lemma proved, the plan is to prove that the added `ASpec` precondition is satisfied using the abstract invariant (as they both relate to the `ASpec`), though this is out of the scope of this thesis as proving invariance across the entire `ASpec` will take time.

5.3.2 Intermediate Step

Directly relating the `ASpec` precondition to the `ExecSpec` assertion was a fairly involved task, requiring several large changes to be made to the steps in the original proof in order to relate the extra precondition and assertion. Whilst this was definitely feasible to complete, splitting the modifications into two separate steps, each proving a separate lemma, made the process simpler.

The final step, as discussed above, was to generate the required correspondence proof with the precondition on the `ASpec` side, but prior to that I proved a separate lemma with the corresponding precondition on the `ExecSpec` side. The shape of this lemma is presented in Snippet 5.13.

Snippet 5.13: Shape of Intermediate `corres` Lemma

```
proposition corres_f_inCurDomain':
    "corres r
        P (P' and (λs. isInDomainColour (ksCurDomain s) ptr))
        (f ptr) (f'' ptr)"
```

This was much simpler to prove, as the added precondition directly proved the assertion true, as they were both on the `ExecSpec` side. As such, the only changes that needed to be made to the proof was renaming the lemmas utilised. These had to be changed to instead use counterpart lemmas for the modified method (which also needed to be proved).

From there, moving the precondition to the `ASpec` side was a two line proof, as there is already existing machinery in L4v to strengthen either of the preconditions based on the other specification's preconditions. In this manner, the process of replicating a given correspondence proof for a modified method boiled down to

1. reproving lemmas about the modified method
2. replacing rules to use these modified lemmas
3. adding a two line proof for each `corres` lemma being replicated

This is a very manageable process to emulate for the remainder of the `ASpec`, even if tedious when repeated across the entire specification.

The only other bit of work required for this proof pattern to work was a proof relating the `ExecSpec` and `ASpec` oracles. As these were both axiomatised, this was also effectively 'axiomatised' by adding a property to the `ExecSpec` oracle stating its definition matched its `ASpec` counterpart. Once the oracles are replaced by a concrete implementation, this property will need to be properly proved, but will only need to be done once for the entire refinement process.

5.3.3 Proofs of Concept

With this pattern of proof, I successfully replicated and modified eight correspondence proofs regarding `get_object` and `set_object` to include the added assertion and precondition. Each method had four lemmas, each accounting for one of the types of objects that `get_object` and `set_object` could be called upon: thread control blocks, page table entries, address space ID pools and all other objects as a catch-all case. The proof pattern successfully worked when completing all of these proofs, including when these proofs had the added complication of depending on each other. The shape of one of these proofs is provided in Snippet 5.14.

Snippet 5.14: `corres` Lemma for modified `get_tcb`

```
lemma corres_get_tcb_inCurDomain:
    "corres
        (tcb_relation ∘ the)
        (tcb_at t and (λs. t ∈ colour_oracle (cur_domain s))) (tcb_at' t)
        (gets (get_tcb t)) (getObjectInCurDomain t)"
```

I also attempted a proof for `get_cap`, a higher level method which Thomas Liang had added assertions to [Liang, 2025]. The statement of this proof is presented in Snippet 5.15.

Snippet 5.15: Potential `corres` Lemma for modified `get_cap`

```
proposition get_cap_inCurDomain_corres_P:
    "corres
        (λx y. cap_relation x (cteCap y) ∧ P x)
        (
            cte_wp_at P cslot_ptr and
            (λs. cte_map cslot_ptr ∈ colour_oracle (cur_domain s))
        )
        (pspace_aligned' and pspace_distinct')
        (get_cap cslot_ptr) (getCTEInCurDomain (cte_map cslot_ptr))"
```

I mostly completed this proof, with the main proof pattern successfully being emulated. However, in the process of proving counterpart lemmas for the modified method, an extra requirement of the oracle was identified. The oracle needed to ensure that the entirety of a TCB was allocated to its security domain. This result relies upon the fact that cache colours are page aligned, which means the oracle's memory allocation must similarly be aligned to pages, with an entire page being allocated to one domain.

Whilst this requirement is self-evidently true based on the definition of colours, the manner in which the oracle was implemented meant that the property needed to be explicitly stated for it. The property was identified as a requirement late in the thesis and was unable to be articulated as a property of the oracle in time for this report. This meant that lemmas showing that the modified method, `getCTEInCurDomain`, did not fail had to be `sorry`-ed.

With those lemmas `sorry`-ed, however, the proof successfully completes, indicating that the pattern does continue to work with higher level methods.

5.4 Reuse of Prior Time Protection Work

The investigation into the feasibility of reusing work from the prior attempting of verifying time protection (up to 2023) was very short-lived. As the invariant approach was selected, the prior TA accounting work became obsolete. This is due to the fact that the invariant already enforces

the condition that all objects in regions of memory accessible to a given security domain should only point to other addresses accessible to a given security domain.

This meant that that all work done in the security domain's time slice can only affect its own partition of memory. Hence, there was no longer any need to track touched addresses using the assertions added previously, as the invariant shows any possibly touched addresses already lie within the correct regions of memory.

As such, this goal was met early in the thesis, with the conclusion being that the TA accounting would not be reusable. Note that this does not include any other parts of the prior time protection work, such as the theories written to make use of existing access and information flow theories to derive properties relevant for time protection. The use of such work was out of the scope of this thesis, and may remain useful in future work on time protection.

Chapter 6

Future Work

We now discuss the outcome of my work discussed in Chapter 5 and how this proof work can be extended to encompass the entire `ASpec`. This covers both the invariant proof work (Section 6.1) and its refinement to the `ExecSpec` (Section 6.2).

6.1 Invariant

6.1.1 Repairing existing proofs

As mentioned in Section 5.2, the articulated invariant underwent several revisions before arriving at the current iteration, which is quantified over pointer, kernel object and domain. The modification of the invariant condition to be across all domains in the kernel's domain list was done relatively late in the thesis, and broke several proofs which had been completed prior. As such, whilst most of these proofs have now been repaired, there remains one major proof to be repaired, which is the proof showing invariance across `transfer_caps_loop`. Any changes made to the lemma during the repair of this proof will likely break other proofs (the obvious example being `transfer_caps`, which is a wrapper function around it), which likely means that there are also a handful of other repairs yet to be made.

This method presented a considerable challenge in its proof previously, and as such the proof was unable to be completed in time for this report. However, the approach for repairing the proof should follow a similar pattern to the repair of other proofs, and is likely resolvable with around a couple of weeks' worth of effort.

6.1.2 Completing Proof of Invariance

Remaining Work

To add the invariant to the abstract invariants in the `ASpec`, we need the invariant to be true across `call_kernel`. As mentioned in Section 5.2, `schedule` and `activate_thread` have had proofs completed for them, completing two of the three functions called by `call_kernel`. The final method, `handle_event`, is a generic event handler which delegates to one of several methods depending on what event it receives. This method now has proofs for a small proportion of its cases, but there remain several more cases to address.

All four of the fault handlers and the one interrupt event handler have yet to be addressed, as well as three of the five system call event handlers. The remaining two system call event handlers are where the majority of the proofs outlined in Section 5.2 have been targeted, building towards a proof of invariance across the shared method which both handlers call, `handle_invocation`. `handle_invocation` is missing proofs for

- `cdl_intent_op`
- `cdl_intent_cap`
- `cdl_intent_extras`
- `lookup_cap_and_slot`
- `lookup_extra_caps`
- `decode_invocation`
- `mark_tcb_intent_error`
- `perform_invocation`
- `restart`
- `corrupt_ipc_buffer`

All of these methods remain uninvestigated and are large methods in their own right, meaning there is still a large amount of work left even within `handle_invocation`. When all of this work is complete, the invariant should end up proven true, enabling refinement to progress to completion as well.

Pattern to Progress

The previously established pattern to proving invariance, namely utilising `crunch` where possible to reduce manual workload and using weakest precondition style proofs for the remaining methods, should remain fruitful with these remaining methods. This is evidenced by the fact that this style of reasoning successfully completed over 45 invariance proofs at varying levels of complexity, ranging from the most fundamental building block methods to the higher level methods called directly by `call_kernel`.

The main difference that will be encountered with the remaining methods is that they are generic event handlers, meaning the proof work required for each method will be larger. For example, `handle_invocation` will have to demonstrate that the nondeterministic state monads which it passes into `syscall` maintain the kernel object invariant condition (as is required by `syscall`'s side conditions for its invariance proof), which will complicate the proof significantly. As such, these event handler proofs will likely require significantly more steps (and time) than the existing proofs.

6.2 Refinement

6.2.1 Updating `ExecSpec`

For the refinement of the kernel object invariant to proceed, a full stack of methods, all incorporating the `isInDomainColour` assertion written by Thomas Liang [Liang, 2025], needs to be written so that concrete correspondence proofs can be used to discharge C code level assertions. This stack can be written in three ways:

- the existing set of methods can be updated by simply updating `getObject` and `setObject`, which are used by all higher level methods and hence will add the relevant assertions everywhere else.
- write a complete copy of the `ExecSpec` methods, which depend on each other, and which use a modified `getObject` and `setObject` with the assertion added at the lowest level. This is effectively the same as the first option, but keeping the original methods the same, and creating a copy of each method instead.
- write a copy of each method which adds one large assertion for itself and uses lower level methods **without** the assertions.

The first option is evidently the simplest in terms of updating the specification as it only entails updating two methods, but it may make the concrete correspondence proofs between the `ExecSpec` and `CSpec` more difficult. This is because the assertions are all at the lowest level with one assertion per memory access, so the number of these assertions may explode when handling higher level methods. The second option is an ineffective combination of the two other options, which has difficulty both in creating a copy of the entire `ExecSpec` **and** in handling the large number of assertions in proofs. The final option requires creating a completely separate copy of all methods in the specification, as well as all lemmas about these methods. This may be difficult to write, but will likely simplify the correspondence proofs. This is because each proof will have to handle one large assertion (as the definition of each modified method utilises unmodified versions of other methods) which is easier to reason about at all levels.

At the moment, Thomas Liang has found that the final option is the simplest for his `ccorres` proofs [Liang, 2025]. It allows him to extend the existing correspondence proofs to prove a modified version of the lemma, rather than having to duplicate the lemma's proof and modify it. Hence, this is likely to be the way that the `ExecSpec` will be updated in future, which will impact how correspondence proofs between the `ASpec` and `ExecSpec` will work.

6.2.2 Completing Correspondence Proofs for `ExecSpec` methods

With the specification itself updated, the proof obligations generated by the added assertions must be satisfied. Naturally, the proofs will get increasingly difficult as the assertion conditions become more complex in higher level methods.

Nonetheless, I expect the proof pattern established in Section 5.3 to continue to be usable for these higher level correspondence proofs, as evidenced by the success in emulating the pattern for the more complex `get_cap`. This should mean that novel proof work should be minimised when proving correspondence across the `ExecSpec`, with most of the work being copying existing proofs and renaming the utilised rules to use the counterpart proofs against the modified method.

There is also the requirement of proving correspondence between the implementation of the oracle in the `ASpec` and `ExecSpec`. This proof will replace the `sorry`-ed lemma which currently relates the two existing oracles. The difficulty in completing this proof will depend on the complexity of the implementation, but importantly it will only need to be proved once: the remaining results can reutilise this proof. As such, the difficulty of this proof is not of great concern.

Chapter 7

Conclusion

μTCs are a type of security vulnerability which exploit the variations in the timing of hardware operations to leak information between processes. As these hardware operations are abstracted away from software, the only way to close these channels is to address them at the kernel level, and this task is of critical importance for security oriented microkernels such as seL4 [Ge, 2019].

In this report, I discussed the existing plan suggested by Buckley et al. to formally verify the absence of μTCs in seL4 through an extension of the existing security proofs in L4v, adding accounting for the set of addresses touched during execution [Buckley et al., 2023]. I then highlighted the magnitude of proof breakages generated by this process, and suggested an alternative pathway which avoids the years worth of proof work required to repair these breakages.

This new approach involved adding an invariant to the `ASpec`, and refining this result down to the `CSpec` through the `ExecSpec`. This central invariant states that any kernel object accessible to a given security domain only points to other memory accessible to the security domain. With this novel approach detailed, I then recounted my progress in proving the invariant across the `ASpec` in Chapter 5, beginning with relevant definitions in Section 5.1, progressing onto the invariant lemmas in Section 5.2 and ending with the proof of concept of refinement down to the `ExecSpec` in Section 5.3.

Finally, I outlined the takeaways from the results of this work in Chapter 6. I discussed the pathway to completing an invariant proof of the new invariant condition across the `ASpec` in Section 6.1, beginning with the repair of proofs broken by a late change to the definition and ending with a discussion of the remaining work required to complete the proofs. Similarly, I discussed the work done by both Thomas Liang and myself in refining this result in Section 6.2, and how the outcomes of the `ASpec` to `ExecSpec` proof of concept should affect future refinement work.

With these takeaways, there is now a clear plan for continuing the work begun in this thesis. This should facilitate proofs verifying cache colour spatial partitioning in the `CSpec` via a refinement step in the `ExecSpec`, where assertions were added [Liang, 2025] and proved by this invariant. In this manner, the spatial partitioning of memory according to cache colour allocations can be

proved in seL4. This satisfies one requirement of time protection and brings us one step closer to proving seL4 is fully secure.

Bibliography

[Arm Limited, 2024] Arm Limited (2024). *Arm Architecture Reference Manual for A-profile architecture*. Arm Limited.

[Backes et al., 2010] Backes, M., Dürmuth, M., Gerling, S., Pinkal, M., and Sporleder, C. (2010). Acoustic side-channel attacks on printers. pages 1–16, Washington, DC.

[Bernstein, 2005] Bernstein, D. J. (2005). Cache-timing attacks on AES.

[Braun et al., 2015] Braun, B. A., Jana, S., and Boneh, D. (2015). Robust and efficient elimination of cache and timing side channels. *arXiv preprint arXiv:1506.00189*.

[Brickell et al., 2006] Brickell, E., Graunke, G., Neve, M., and Seifert, J.-P. (2006). Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive*, 2006:52.

[Brumley and Boneh, 2003] Brumley, D. and Boneh, D. (2003). Remote timing attacks are practical. pages 1–14, Washington, DC, US.

[Buckley and Sison, 2023] Buckley, S. and Sison, R. (2023). Time protection handover, 07/2023. `https://github.com/au-ts/l4v-private/blob/experimental-timeprot/proof/infoflow/timeprotection/timeprot-handover_2023-07.md`. Last accessed 17 November 2025.

[Buckley et al., 2023] Buckley, S., Sison, R., Wistoff, N., Millar, C., Murray, T., Klein, G., and Heiser, G. (2023). Proving the absence of microarchitectural timing channels. *arXiv preprint arXiv:2310.17046*.

[Cock et al., 2008] Cock, D., Klein, G., and Sewell, T. (2008). Secure microkernels, state monads and scalable refinement. pages 167–182, Montreal, Canada.

[de Roever and Engelhardt, 1998] de Roever, W.-P. and Engelhardt, K. (1998). *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Number 47. United Kingdom.

[Dijkstra, 1975] Dijkstra, E. W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457.

[Ge, 2019] Ge, Q. (2019). *Principled Elimination of Microarchitectural Timing Channels through Operating-System Enforced Time Protection*. PhD thesis.

[Ge et al., 2018] Ge, Q., Yarom, Y., Cock, D., and Heiser, G. (2018). A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. 8:1–27.

[Genkin et al., 2015] Genkin, D., Pachmanov, L., Pipman, I., and Tromer, E. (2015). Stealing keys from PCs using a radio: Cheap electromagnetic attacks on windowed exponentiation. pages 207–228, Saint Malo, FR.

[Genkin et al., 2014] Genkin, D., Shamir, A., and Tromer, E. (2014). RSA key extraction via low-bandwidth acoustic cryptanalysis. pages 444–461, Santa Barbara, CA, US.

[Heiser, 2020] Heiser, G. (2020). The seL4 microkernel – an introduction. seL4 Foundation Whitepaper.

[Heiser and Elphinstone, 2016] Heiser, G. and Elphinstone, K. (2016). L4 microkernels: The lessons from 20 years of research and deployment. *ACM Transactions on Computer Systems*, 34(1):1:1–1:29.

[Hoare, 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580.

[Klein et al., 2014] Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., and Heiser, G. (2014). Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70.

[Klein et al., 2009] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. (2009). seL4: Formal verification of an OS kernel. pages 207–220, Big Sky, MT, USA.

[Klein et al., 2010] Klein, G., Sewell, T., and Winwood, S. (2010). *Refinement in the formal verification of seL4*, pages 323–339.

[Kocher et al., 2019] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Haburg, M., Lipp, M., Mangard, S., Prescher, T., Schwartz, M., and Yarom, Y. (2019). Spectre attacks: Exploiting speculative execution. pages 19–37, San Francisco, CA, US.

[Kocher et al., 1999] Kocher, P., Jaffe, J., and Jun, B. (1999). Differential power analysis. volume 1666, pages 388–397.

[Lampson, 1973] Lampson, B. W. (1973). A note on the confinement problem. *Communications of the ACM*, 16:613–615.

[Liang, 2025] Liang, T. (2025). Refining seL4's accounting of touched addresses for time protection. An Unpublished Honours Thesis Report.

[Lipp et al., 2018] Lipp, M., Schwartz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. (2018). Meltdown: Reading kernel memory from user space. Baltimore, MD, USA.

[Liu et al., 2016] Liu, F., Ge, Q., Yarom, Y., Mckeen, F., Rozas, C., Heiser, G., and Lee, R. B. (2016). CATalyst: Defeating last-level cache side channel attacks in cloud computing. pages 406–418, Barcelona, Spain.

[Masti et al., 2015] Masti, R. J., Rai, D., Ranganathan, A., Müller, C., Thiele, L., and Čapkun, S. (2015). Thermal covert channels on multi-core platforms. pages 865–880, Washington, DC, US.

[Murdoch, 2006] Murdoch, S. J. (2006). Hot or not: revealing hidden services by their clock skew. pages 27–36, Alexandria, VA, US.

[Murray et al., 2013] Murray, T., Matichuk, D., Brassil, M., Gammie, P., Bourke, T., Seefried, S., Lewis, C., Gao, X., and Klein, G. (2013). seL4: from general purpose to a proof of information flow enforcement. pages 415–429, San Francisco, CA.

[Nipkow et al., 2002] Nipkow, T., Paulson, L., and Wenzel, M. (2002). *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283.

[Osvik et al., 2006] Osvik, D. A., Shamir, A., and Tromer, E. (2006). Cache attacks and countermeasures: The case of AES. pages 1–20, San Jose, CA, US.

[Paulson, 2013] Paulson, L. C. (2013). Isabelle's logics: FOL and ZF. `http://isabelle.in.tum.de/doc/logics-ZF.pdf`.

[Quisquater and Samyde, 2001] Quisquater, J.-J. and Samyde, D. (2001). Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. pages 200–210, Cannes, FR.

[Schaefer et al., 1977] Schaefer, M., Gold, B., Linde, R., and Scheid, J. (1977). Program confinement in KVM/370. pages 404–410, Atlanta, GA, US.

[Trustworthy Systems Team, 2017] Trustworthy Systems Team, D. (2017). *seL4 Reference Manual, Version 7.0.0*.

[Yarom and Falkner, 2014] Yarom, Y. and Falkner, K. (2014). FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. pages 719–732, San Diego, CA, US.