Formal Verification of Capability-Safety in the CHERIoT-Ibex Processor

Tita Rosemeyer

University of Oxford

A thesis submitted for the degree of

MSc in Mathematics and Foundations of Computer Science

Trinity Term 2025

Acknowledgements

Words cannot express my gratitude to my supervisor, Tom Melham, whose kindness, support and belief in me have been constant throughout this project. His willingness to devote time to detailed feedback, engaging discussions and patient guidance has been invaluable to this work. I could not have hoped for a better supervisor.

This project would not have been possible without the brilliant people at SCI Semiconductor. I am especially grateful to David Chisnall, Nathaniel Wes Filardo and Robert Norton, for their patience, expertise, and willingness to answer even my most naive questions. Their generosity with their time and willingness to help me work through difficulties, gave me the confidence to keep going. Special thanks as well to Louis-Emile Ploix for his helpful insights and tips on formal verification of the CHERIoT-Ibex processor.

Finally, my deepest thanks go to my family and friends. To those who bravely asked, 'So, what's your thesis about again?' — and listened through the whole answer. To my Dad and Theo, who made sure I actually got to work, to Mimi and Roger, who made sure I didn't work all the time; and to my Mom, Carola, and Jenny, for providing me with the environment to do both. Finally, to Lieselotte and Hans for keeping my inner child alive. (According to Lieschen, I'm not a grown-up anyway: case closed.)

Abstract

This dissertation presents a comprehensive formal verification of certain key capability safety properties in the unsealing mechanism of a real-time operating system for the CHERIoT-Ibex processor — a RISC-V core extended with the CHERI architecture. Some pathfinding results for the context-switching mechanism are also presented. We explore a new approach to reasoning about such low-level code: applying hardware model checking techniques to prove safety and non-interference properties of binary machine code running on a formal model of the processor RTL. The properties are expressed in Linear Temporal Logic (LTL). In general, LTL model checking is co-NP complete, but we achieve practical scalability using the k-induction algorithm combined with modular decompositions into smaller lemmas, resulting in experimentally linear time complexity.

CHERI introduces capabilities: unforgeable, hardware-enforced memory references that combine memory addresses with meta-data about bounds and permissions, enabling fine-grained memory protection and software compartmentalisation. In multitasking systems, context switches must ensure that capabilities belonging to one thread or process are not accessible by another. Similarly, the unsealing mechanism — which handles sensitive capabilities — must not expose these during execution.

This work formally verifies that the unsealing mechanism, implemented in low-level software, correctly enforces capability-safety: no sensitive capabilities, or capabilities derived from these, are leaked. It also establishes partial verification results for the context-switching mechanism, showing that derived capabilities from one thread or process are not accessible after a switch. These key properties of the lowest level of the operating system are vital for security, and were identified as being of interest and value by the industrial collaborators on this project, SCI Semiconductor.

Our novel approach to verification formalises the execution of binary code (implementing unsealing and context switching) running directly on a cycle-accurate, formally verified RTL model of the processor. To the best of our knowledge, this is the first project to formally verify binary code directly executing on formally verified hardware. Other approaches typically reason at the C or C++ level, leaving a substantial compilation gap between what is verified and what is actually executed in fielded systems. At best, existing work verifies assembly code running on a formal, and possibly abstracted, model of the ISA semantics —and not the actual processor architecture. Our approach eliminates this gap, enabling end-to-end verification from hardware to low-level software, offering unusually strong guarantees for capability-safety.

Introduction

Low-level software, including real-time operating systems (RTOS), bootloaders and firmware, forms the foundation of systems execution and security. These components run at the highest privilege levels and control access to critical hardware resources. A compromise at this level can undermine the entire system's integrity and malicious attacks often target the lower-level components to evade application-level protections and gain control of entire devices [1]. As such components are increasingly used in connected and safety-critical environments — including aviation, industrial control, medical devices and the Internet of Things (IoT) — the consequences of low-level vulnerabilities can extend to significant societal and safety risk.

Despite its criticality, low-level software is notoriously difficult to secure. It is often implemented in memory unsafe languages, such as C or assembly, making it highly susceptible to classes of bugs, such as buffer overflows, use-after-free errors and null pointer dereferences which are all mentioned in CWE's 25 most dangerous software weaknesses 2024 [2]. Moreover, such software often employs concurrency, interrupts and complex control-flows — features that complicate both reasoning and analysis. Deployment contexts exacerbate these challenges: components may run in remote or physically inaccessible systems such as satellites [3], medical implants, or industrial machinery. In these scenarios, logging and debugging capabilities are limited or non-existent and updating software to patch discovered vulnerabilities may be infeasible. While software testing remains a common practice, it cannot exhaustively explore all execution paths, leaving room for undetected bugs in corner cases. This makes computer-aided *formal verification* methods an essential step towards achieving strong assurance in these critical components.

Model checking techniques. Formal verification uses mathematical methods to check whether a system satisfies given correctness or security properties. Main ap-

proaches include abstract static analysis, model checking, and symbolic simulation [4]. *Model checking* is one of the most widely used techniques. It works by exhaustively exploring all reachable states of the system model to determine whether a given property — often specified in temporal logics such as Linear Temporal Logic (LTL) — holds. If the state space is finite, model checking is guaranteed to terminate with either a proof or a counterexample, but the computational complexity is high. In practice, scalability is achieved through mathematical techniques such as k-induction, which applies the induction principle to establish unbounded guarantees with reduced complexity. Another widely used approach is bounded model checking, which restricts exploration to a fixed depth. While it cannot provide full proofs, it is effective for detecting counterexamples within the chosen bound. *Abstract static analysis* constructs a sound, over-approximated model of the system's behavior, allowing properties to be proved without full state enumeration — but at the cost of possible imprecision (e.g. spurious alarms). *Symbolic simulation* provides symbolic input values and simulates their evolution through the system execution to obtain expressions of relevant signals in terms of those input values.

Previous Approaches. Most formal verification of low-level software operates at the source code level — typically C or C++. Tools such as CBMC, Java PathFinder, and SATABS support automated model checking and symbolic reasoning over these high-level programs [4]. But this introduces a substantial abstraction gap between what is verified and what is actually executed in fielded systems: verified source code must still pass through compilers, linkers and hardware-dependent runtime layers before execution. The correctness of those layers is often assumed, but not verified. As noted in the seL4 microkernel verification effort [5]: 'We assume the correctness of the compiler, assembly code, boot code, management of caches, and the hardware; we prove everything else.' Existing work that verifies lower level assembly code typically relies on a formal, and possibly abstracted, model of the ISA semantics — and not the actual hardware implementation. This means that compiler bugs, hardware anomalies, or mismatches between specification and silicon remain unverified. This

project explores a new verification approach that verifies the execution of compiled binary code running directly on a cycle-accurate formally verified RTL model of the target processor. This approach closes the gap between high-level correctness guarantees and real-world execution, enabling end-to-end assurance for capability-safety properties in the firmware.

CHERIoT. To demonstrate this verification approach, we target CHERIoT, a capability-based security architecture designed for resource-constrained IoT systems [6]. This project focuses on verifying the unsealing mechanism of the CHERIoT RTOS for the CHERIoT-Ibex processor [7]. The processor and RTOS were designed from the ground up to enforce object-granularity memory safety on an embedded microcontroller [8], making it an ideal case study for verifying low-level security-critical behavior. Both the CHERIoT RTOS and the CHERIoT-Ibex RTL implementation (in SystemVerilog) are publicly available [7, 9].

CHERI (Capability Hardware Enhanced RISC Instructions) [10] is a hardware-software architecture that introduces *capabilities*: unforgeable, hardware-enforced memory references that combine memory addresses with meta-data about bounds and permissions, enabling fine-grained memory protection and software compartmentalization. CHERIoT (Capability Hardware Extension to RISC-V for Internet of Things) [10] extends the CHERI principles to resource-constrained systems by implementing a capability-aware Instruction Set Architecture (ISA) as an extension to RISC-V. The CHERIoT-Ibex processor, developed by lowRISC C.I.C. and Microsoft, implements this ISA on a 32-bit RV32IMCB core. The custom CHERIoT RTOS running on the processor is being jointly developed by SCI Semiconductor, Microsoft, and Google [9].

Verification targets. The key system components include the *switcher*, the *scheduler* and the *allocator*. This project is focuses primarily on the allocator, which is responsible for managing dynamic memory and provides use-after-free prevention. As a trusted entity — holding a capability with access to the entire heap — its security

and safety is crucial to the safety of the system. Within the allocator, the *unsealer* handles *sealed* capabilities — used to build type-safe, opaque types that remain secure even under mutual distrust and delegation [11]. Critical safety properties of the unsealer include preventing the leakage of the unsealing authority and ensuring that unsealing does not expose sensitive information of the sealed capability. Another verification target is the switcher, the most privileged entity in the system, which performs context switchers. It must ensure strict isolation of thread-local capabilities to prevent accidental or malicious cross-thread access during task transitions.

These key properties of the lowest level of the operating system are vital for security and were identified as being of interest and value by the industrial collaborators on this project, SCI Semiconductor [12].

Contributions. This project's novel contribution is the design of a model checking framework that enables verification of binary code executing directly on a cycle-accurate, formally verified hardware model. To the best of our knowledge, no previous work has addressed verification at this low level. Specifically, our contributions include

- the **logical formulation of code execution** in linear temporal logic (LTL), which required managing complex pipelined processor timing and identifying precise processor signals to capture cycle-accurate instruction execution. This enables direct reasoning about binary code execution through processor RTL signals without compilation abstractions;
- the **formalization of safety properties** — such as leakage constraints and hierarchical ordering of capabilities — in LTL, requiring novel techniques for capability derivation relationships and register file scanning with state capture. This enables automated verification of memory security properties previously verified only manually;
- an **instantiation of this framework** on the *unsealer* module of the CHERIoT RTOS allocator, including a case study and experimental evaluation demonstrating scalability and approximate linearity with program size, demonstrating

the practical applicability of the approach;

- the discovery of a **previously unknown bug** in the unsealer logic, where the unsealed object pointer was leaked during exception handling [13]

Together, these results provide an initial demonstration of the technical feasibility of mathematical verification of binary code running directly on its target processor. This is verification at an unprecedented level of fidelity to fielded systems, eliminating the gaps and modelling abstractions discussed earlier.

Tools. Proofs were carried out using Jasper, a commercial formal verification tool by Cadence Design Systems [14] that supports formal property checking and symbolic analysis. In our novel approach, the formal RTL model within Jasper serves as a definitive formal semantics of code execution for binary-level software verification. Both the CHERIoT RTOS (including the context-switching code) and the CHERIoT-Ibex RTL implementation (in SystemVerilog) are publicly available [9, 7]. All code, properties and scripts developed for this thesis are available in a public repository [15].

Contents

Chapter 1

Background

We assume the reader's familiarity with:

- propositional logic and satisfiability,
- state transition systems and Kripke structures, and
- Linear Temporal Logic (LTL).

In this thesis, we only rely on the safety fragments of LTL, i.e. properties using the operators **X** ('next') and **G** ('globally'). No liveness operators are used, which is typical for processor verification.

1.1 Model checking

Baier and Katoen define *Model Checking* in their book *Principles of Model Checking* (2008) [16] as follows:

> Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.

Model checking techniques. In practice, model checking algorithms fall broadly in two categories. *Explicit-state techniques* explore the state space directly, enumerating reachable states one by one, but often suffer from the *state explosion problem*, where the number of reachable states grows exponentially with system variables. *Symbolic techniques*, by contract, represent sets of states and transitions compactly, for example using Binary Decision Diagrams (BDDs) or satisfiability (SAT/SMT) encodings. Modern practice is dominated by symbolic methods, and this thesis uses

exclusively SAT-based model checking, which leverages advances in propositional satisfiability solvers to scale verification to realistic hardware/software systems.

SAT-based Model Checking. In SAT-based model checking, the verification problem is encoded as a propositional satisfiability instance. The system is modelled as a finite-state transition system M and the safety property as an LTL formula φ. Given these, a Boolean formula $F(M, \neg\varphi)$ is constructed such that:

$$F(M, \neg\varphi) \text{ is satisfiable } \iff M \not\models \varphi$$

In other words, the SAT solver checks to see if a counterexample to the safety property — a valuation satisfying $F(M, \neg\varphi)$ — can be constructed. If the formula is satisfiable, its witness constitutes a counterexample disproving the property; otherwise, the property is proven.

Relevant techniques. Two relevant SAT-based techniques are presented below. The first technique, *bounded model checking (BMC)*, restricts the search for counterexamples to a given depth bound k. The second technique, *k-induction*, uses BMC to check whether the property can be proven by induction, thus providing unbounded guarantees. The explanations of these techniques are largely taken from chapter 10 of *Model Checking, second edition* by Kroening et al. (2018) [17].

Bounded model checking

Bounded model checking (BMC) [17] constitutes the unravelling of the system M and formula φ up to a given depth bound k. Given a transition system M and a property φ, BMC constructs a propositional formula that is satisfiable if and only if there exists a path of length k in M that violates φ — a counterexample of length at most k. If the formula is unsatisfiable, there exists no counterexample of length k or less, but there might exist one that is longer.

Thus, BMC is *bug-finding oriented*: it is exceptionally good at uncovering shallow counterexamples but does not by itself guarantee correctness unless the bound k exceeds a previously proven completeness threshold k_{max}, i.e., a bound such that if no counterexample exists of length $\leq k_{max}$, then no counterexample exists of any length and the property holds universally.

Encoding Strategy. Let a finite-state transition system be represented as a Kripke structure:

$$M = (S, S_0, R, AP, L)$$

where:

- S is the finite set of states,
- $S_0 \subseteq S$ is the set of initial states,
- $R \subseteq S \times S$ is the transition relation,
- AP is the set of atomic propositions,
- $L : S \to 2^{AP}$ is a labelling function.

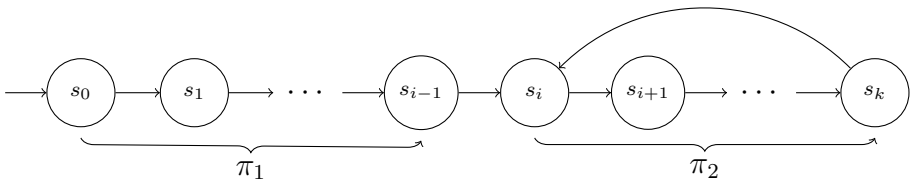
We can also assume that we can represent S_0, R and any $p \in AP$ as first-order predicates over the states of M. For example, we write $p(s)$ for an atomic proposition $p \in AP$ and state $s \in S$ to mean that L labels s with p. Each state $s \in S$ is encoded as a valuation of Boolean variables $V = \{v_1, \ldots, v_n\}$. To represent a path of length k, we introduce $k+1$ copies of these variables: $V_0, V_1, \ldots, V_k$. A tuple of valuations $(s_0, \ldots, s_k)$ of these copies corresponds to a candidate path.

Path formula. We can now construct a boolean formula $path_k$, that holds true if and only if $\pi = s_0, \ldots, s_k$ is a path of M:

$$path_k(s_0, \ldots, s_k) \iff S_0(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \tag{1.1}$$

This enforces that s_0 is an initial state and every adjacent pair respects the transition relation. This definition can be extended to infinite paths, by extending the tuple $(s_0, \ldots, s_k)$ into an infinite sequence $(s_i)_{i \in \mathbb{N}}$ and requiring $S_0(s_0)$ and $R(s_i, s_{i+1})$ for any $i \in \mathbb{N}$.

Lassos. Some temporal properties can be refuted by finite counterexamples, but others require infinite executions. For example, the property $\mathbf{F}p$ ('p eventually holds in any infinite path') can only be disproved by an infinite counterexample. Such infinite counterexamples are represented in practice by *lassos*: paths consisting of a finite prefix π_1 (the *stem*) followed by a finite cycle π_2 (the *loop*) that repeats indefinitely:



If the states along the lasso are $s_0, \ldots, s_k$, we call it a lasso of length k, in the same way that a finite path of k transitions is described by its $k+1$ states. A lasso is a counterexample of $\mathbf{F}p$ if all of the states in its stem or loop satisfy $\neg p$. Intuitively, the execution 'loops back' to a previously visited state and cycles forever without encountering p. Formally, a lasso of length k can be expressed as:

$$lasso_k(s_0, \ldots, s_k) \iff S_0(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k-1} s_k = s_i \tag{1.2}$$

where the first part mirrors the path equation (1.1), and the final condition ensures that the last state s_k coincides with some earlier state s_i, forming the loop

Tableaus. For the construction of a BMC encoding for arbitrary LTL properties, we introduce Kripke structures with fairness, called *tableau*s. A finite-state Kripke structure with fairness extends the 5-tuple (S, S_0, R, AP, L) of the Kripke structure (see above) by a set F of fairness constraints, where $F = \{P_1, \ldots, P_n\}$ with $P_j \subseteq S$. In this Kripke structure, a path is called *fair* if it visits each of the sets P_i in F infinitely often. Given any LTL formula φ, we can construct a corresponding tableau T_φ such that any infinite path in T_φ is fair if and only if it satisfies φ. While not all fair paths will have the shape of a lasso, it is true that there is a fair path on T_φ if and only if there is a fair lasso-shaped path on T_φ, so we can restrict to lasso shaped paths in the following search for counterexamples.

BMC translation. Let's suppose we are given an LTL property φ and system model M (as a Kripke structure), and want to check whether $M \models \varphi$. To do this, we construct the tableau $T_{\neg\varphi}$ which accepts exactly the paths violating φ and then form the product of M with $T_{\neg\varphi}$. Let $\Psi = (S^\Psi, S_0^\Psi, R^\Psi, AP^\Psi, L^\Psi, F^\Psi)$ be the resulting structure. The product construction ensures that paths in the combined structure simultaneously respect the dynamics of M and the temporal restrictions of $T_{\neg\varphi}$. Any fair path of Ψ is thus (1) corresponding to a computation of M and (2) satisfies $\neg\varphi$, and therefore constitutes a counterexample to φ on M. Thus, there exists a counterexample to φ on M if and only if there exists a fair path on Ψ and thus if and only if there exists a fair, lasso-shaped path on Ψ.

Formula construction. We now construct a formula that is satisfiable if and only if there is a fair, lasso-shaped path on Ψ of length k:

$$S_0^\Psi(s_0) \wedge \bigwedge_{i=0}^{k-1} R^\Psi(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k-1} \left(s_k = s_i \wedge \bigwedge_{P \in F^\Psi} Fair_i^P \right) \tag{1.3}$$

This is the lasso equation (1.2) with the additional fairness condition in the final conjunct ensuring that each fairness constraint is satisfied in the loop of the lasso path. The fairness condition $Fair_i^P$ for $P \in F^\Psi$ encodes that the loop starting from state s_i visits P and is encoded as follows:

$$Fair_i^P \iff \bigvee_{j=i}^{k-1} P(s_j)$$

The formula (1.3) can then be passed to a SAT solver to check for satisfiability. If it is satisfiable, a counterexample to the property φ in M has been found. If it is not satisfiable, no counterexample of length smaller or equal to k exists, but there might be one that is longer.

k-Induction

If aimed at verification (providing guarantees of safety properties), BMC relies on unwinding the model and property up to bound k that exceeds a previously computed and proven completeness threshold. Since the threshold is typically proportional to the system diameter, and often infeasible to compute, BMC alone is often insufficient in establishing unbounded correctness of temporal properties. We now introduce a technique that employs unwindings as a building block, leveraging the mathematical principle of induction to prove properties using only a few unwinding. We will explain the technique on the example of a simple global property Gp, where p is a state predicate.

Mathematical Foundation. The k-induction technique [17] builds directly on the classical induction principle which is used to prove that a property holds for all natural numbers. This is done in two steps: first, one proves that the property holds for the smallest number (typically 0 or 1); second, one shows that whenever the property holds for a given number $n-1$, it also holds for n. The induction principle then guarantees that the property holds for all natural numbers. Formally, given a predicate $Q(n)$ over the natural numbers, to prove $\forall n Q(n)$ one establishes:

1. **Base case**: $Q(0)$.
2. **Inductive step**: $Q(n-1) \rightarrow Q(n)$ for all $n >$.

We will apply this principle to invariance properties of the form $\mathbf{G}p$, which assert that a (desirable) state predicate $p(s)$ holds in all reachable states. Given a system in form of a Kripke structure as above (with S_0 being the initial state predicate and R the transition predicate), the initial states play the role of the number 0, while R plays the role of the successor function $n \mapsto n+1$. Thus, the induction principle specializes as follows:

- **Base case**: show that no initial state violates the property, i.e.:

$$S_0(s_0) \wedge \neg p(s_0) \quad \text{is unsatisfiable}$$

This ensures that the system does not start in a 'bad' state.

- **Inductive step**: show that the property is preserved under transitions, i.e.:

$$p(s) \wedge R(s, s') \wedge \neg p(s') \quad \text{is unsatisfiable}$$

This ensures that once a state satisfies p, every immediate successor does as well. Note that no assumption is made about s being reachable.

In practice, both conditions can passed to a SAT solver for automatic checking. If both are unsatisfiable, then $\mathbf{G}p$ is established.

Generalization to k-induction. Intuitively, k-induction extends the classic induction principle by looking at a block of k past cases instead of just one. This is useful for properties that depend on a history rather than only the immediately preceding state. For example, proving that the Fibonacci sequence is monotonic non-decreasing requires 2-induction: one must assume both $F_{n-2} \geq F_{n-3}$ and $F_{n-1} \geq F_{n-2}$ to conclude $F_n \geq F_{n-1}$. Formally, mathematical k-induction reads:

- **k-induction base case**: $Q(0) \wedge \cdots \wedge Q(k-1)$
- **k-induction inductive step**: $(Q(n-k) \wedge \cdots \wedge Q(n-1)) \rightarrow Q(n)$

In the setting of transition systems, this changes what objects we consider. Plain induction reasons about individual states: if p holds in one state, then it must also hold in its successor. k-induction reasons about paths: we assume that p holds throughout a (not necessarily reachable) sequence of k consecutive states, and then show that it must also hold in the next state. The k-induction base case checks that the property holds throughout any given a path of length $< k$ (starting from an initial state); each state s_i plays the role of the natural number i. Together, these conditions mirror the logic of mathematical k-induction and establish that the property p holds in all states of all reachable paths (which is equivalent to all reachable states). Formally, we write:

- **k-induction base case**: show that no state reachable by a path of length $< k$ violates the property, i.e.:

$$S_0(s_0) \wedge \bigwedge_{i=0}^{k-2} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k-1} \neg p(s_i) \quad \text{is unsatisfiable} \tag{1.4}$$

- **k-induction inductive step**: assume p holds along a sequence of k states, then show it must also hold at the successor, i.e.:

$$\bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigwedge_{i=0}^{k-1} p(s_i) \wedge \neg p(s_k) \quad \text{is unsatisfiable} \tag{1.5}$$

Again, s_0 is not required to be an initial state in the inductive step — any sequence of states consistent with R may serve as the induction hypothesis. This means that if the inductive step equation is satisfiable, the property has not been disproved; a witness could lie in the unreachable state-space and thus not constitute a valid counterexample.

Completeness issues. The k-induction method as described here is not complete; there are properties that cannot be proven with the technique. For example, an unreachable cycle violating the property may block the proof, even though the property holds in all reachable states. To address this, one augments the encoding of the inductive step with a *loop-freedom constraint* ensuring that paths considered in the step case are simple (do not visit the same state twice):

$$\bigwedge_{i=0}^{k-1} \bigwedge_{j=i+1}^{k} s_i \neq s_j$$

With loop-freedom, a completeness result holds: if the property is invariant and k exceeds the completeness threshold, then k-induction proves the property. In practice, many properties are established with k values well below this bound, yielding efficient proofs beyond the capability of plain BMC.

1.2 CHERI

Capability systems

A *capability system* replaces or augments address pointers with *capabilities*: unforgeable tokens of authority that reference objects together with the rights to perform particular operations on them. Intuitively, a capability is a handle that carries its own access control policy; possession of the token *is* the authority. This differs from traditional access-control lists: authority is expressed by distribution of tokens, not by central tables checked at dereference time. The CHERI (Capability Hardware Enhanced RISC Instructions) project [18] realizes capabilities as a first-class architectural value (hardware data-type) rather than a purely software convention. They typically include both a reference to a memory address and metadata that constrains the accessible memory region and operations permitted. This enables architectural enforcement of spatial safety, as well as fine-grained compartmentalization.

Core Principles. CHERI's design is guided by two fundamental principles aimed at minimizing risk and ensuring that authority is exercised safely and deliberately [8]:

1. The principle of **least privilege**: Each component, process, or user should operate with the minimal authority necessary to perform its function.
2. The principle of **intentional use**: Access and actions should occur only as the result of explicit, deliberate requests by the user or system.

These principles shape CHERI's protection model, ensuring that authority is both restricted and used intentionally. Building on them, the core security properties of a capability system implement and enforce these guarantees in practice:

- **Unforgeability**: capabilities cannot be synthesized; authority only arises from valid derivation or creation operations.
- **Monotonicity of derivation & delegation**: capabilities may be derived only to with monotonic non-increase of authority (e.g. shrink bounds, remove permissions, or preserve the same rights while changing the address) and can be handed to other principals to effect delegation.
- **Revocation / temporal safety**: systems provide mechanisms (hardware or software) to prevent reuse of authority after an object is deallocated. CHERI, in its current form, provides strong hardware-enforced spatial safety but leaves temporal safety to software-level mechanisms.
- **Non-bypassability**: all accesses to an object must be mediated by a valid capability; integer addresses alone do not grant authority.

Capability systems are effective for memory safety because they couple spatial bounds, permissions and provenance to pointers themselves; checks are local and can be enforced at point of use. Foundational literature [19, 20] frames these ideas; CHERI implements them in hardware for performance and stronger guarantees [10].

Capability systems and hybrid designs. A *capability system* enforces that all accesses are represented and mediated exclusively through capabilities. By contrast, many conventional systems merely *support* capabilities, meaning that some but not all accesses are capability-mediated. CHERI exemplifies this hybrid approach: it integrates capability support into a conventional RISC architecture and operating systems without requiring that all code use capabilities, and it offers both *pure-capability Application-Binary-Interfaces (ABIs)* — in which all pointers are capabilities — and *hybrid ABIs* — in which conventional pointers and capabilities coexist. Hybrid approaches improve adaptability by enabling incremental migration to least-privilege

programming, although they provide weaker robustness than a fully capability-based environment [21].

CHERI architecture

CHERI's key architectural idea is to make capabilities a first-class hardware type: capability registers, capability-aware instructions, and *tagged memory* that preserves the distinction between ordinary data and capabilities. Enforcing capability invariants in microarchitecture (and in some implementations, dedicated coprocessors) prevents the usual classes of pointer corruption exploits (out-of-bounds access, use-after-free, arbitrary code jumps) while retaining C-language performance and semantics where possible [18, 22].

Metadata encoding. In CHERI, the architectural capability contains structured metadata beyond a raw address. A common, concrete encoding (CHERI ISAv8 / CHERIoT variants) includes:

1. **address field**: the reference to the memory object as a 32- or 64-bit memory address.
2. **bounds**: (logical) base and top bounds (or compressed encodings thereof) that define the contiguous memory region accessible through the capability; all accesses must lie in $[base, top)$.
3. **permissions field**: a set of bits encoding the allowed operations (e.g. Load, Store, Execute, Seal/Unseal, system-register access).
4. **object type / sealing field**: an identifier used for sealing / unsealing semantics (see below).
5. **validity tag**: an integrity bit indicating that the machine treats the word as a capability (tagged); tags are stored and checked by the tagged-memory mechanism and cannot be forged by ordinary store/load sequences. The tag is preserved by capability-aware load/store primitives.

The exact bit-field layout can be found in the CHERI ISA Specification [23]; CHERIoT uses a compact encoding adapted for RV32 [6], which will be explained in detail in Section 2.1.

Essential architectural components of CHERI include: [10, 22]

- **Capability registers and instruction extensions**: CHERI adds a set of capability registers (distinct from general purpose registers) and instructions

for loading/storing capabilities, deriving new capabilities, setting bounds, and checking permissions. Capability instructions preserve the tag bit if used correctly and enforce checks atomically where required.

- **Tagged memory architecture**: Memory stores carry an out-of-band tag state (in hardware or a protected memory structure) so that the processor can distinguish capability words from untagged data. This prevents an attacker from constructing a valid capability by writing raw bytes into memory: an ordinary store clears the tag bit of any memory region it spans. CHERI's tagged memory and enforcing load/store semantics ensure that in-memory capabilities retain integrity.
- **Coprocessor / MMU integration and capability addressing**: CHERI performs capability checks on the addresses generated by capability use. On systems with a memory management unit (MMU) or memory protection unit (MPU), these addresses may then be subject to further translation or protection checks. In this way, a capability encodes the permissible address region, and hardware ensures access only when capability checks succeed.

CHERI capability operations and semantics

CHERI instructions for manipulation of capabilities fall into three broad categories, corresponding to how they transform capability values: [23]

1. **Transfer operations**: *Load* and *store capability* instructions (`(C)LC` , `(C)SC`) move capabilities between registers — preserving the tag bit and performing alignment / tag checks. In addition, when using transitive permissions such as load-mutable or load-global, load operations may also strip or reduce permissions according to the capability's authority rules. When a stored word carries a tag, hardware ensures only capability-aware store semantics may set tag state.
2. **Restriction operations**: Instructions such as `CSetAddr(c,a)` , `CAndPerms(c,p')` , and `CSetBounds(c,b,t)` derive a new capability from an existing one. These are *monotonic* operations: the derived capability grants no more authority than the source (bounds shrink, permissions are restricted).
3. **Transformation operations**: *Sealing* and *unsealing* operations change whether a capability is opaque or usable. The `CSeal(c, otype)` instruction marks a capability as *sealed*, associating it with an object type identifier `otype` . A sealed capability is opaque: it cannot be dereferenced for memory operations.

The `CUnseal(c,ck)` instruction reverses this transformation, but only if the caller holds a capability `ck` with the matching *permit-unseal* authority for the given `otype`. These operations are the basis for the implementation of encapsulation, domain isolation and controlled privilege transitions (see below).

All memory accesses are mediated by capabilities according to the non-bypassability predicate above. In addition, capabilities are subject to immediate faulting if checks fail; the fault semantics are architecturally specified (trap vectors, register state guarantees) so formal models can treat faults as well-defined control transitions.

Sealed capabilities. CHERI introduces a sealing mechanism that extends capabilities to support secure compartmentalization and controlled domain transitions. Sealed capabilities are immutable (in the sense that any manipulation clears the tag bit), non-dereferencable, and protected by hardware against modification or control transfer, making them opaque handles that can be safely passed between mutually distrusting components. This enables their use as secure references for domain transitions, where sealed code and data capabilities together represent complete object references.

Object types. The security of sealed capabilities is enforced through object types, metadata that links related capabilities and governs unsealing. Object types are set during sealing using an authorizing capability, allowing delegation of type-space usage without privileged intervention. Unsealing requires possession of a capability with matching authorization, ensuring that only intended components regain mutable and dereferencable access. In practice, domain transitions are supported by *sealed entry capabilities (sentries)*, code capabilities that are automatically unsealed when used as jump targets, providing the foundation for secure privilege transitions and compartmentalized execution [23, 11].

Chapter 2

Driving Example: CHERIoT

The Capability Hardware Extension to RISC-V for IoT (CHERIoT) platform is a scaled, CHERI-inspired capability architecture tailored for 32-bit RISC-V micro-controllers in IoT and embedded contexts. Drawing upon CHERI principles of spatial and temporal memory safety and compartmentalization, CHERIoT targets radically resource-constrained environments — micro-controllers with limited compute, memory, and power budgets [24].

2.1 CHERIoT-Ibex

The CHERIoT-Ibex is the reference implementation: an extension of the lowRISC Ibex RV32IMCB (a 32-bit RISC-V micro-controller) incorporating CHERIoT ISA features. It can be configured with either a 2-stage or 3-stage pipeline, similar in structure to the baseline Ibex core. The CHERI functionalities are optional and can be configured via the flag `CHERIoTEN` [7].

Extensions to Ibex. Integration of CHERIoT capabilities into the Ibex core ensures that capability instructions, tags and safety mechanisms are deeply embedded in the micro-architecture rather than bolted on at software level [7]:

- The **register file** is modified so that the original 32-bit general-purpose registers are extended to 65-bit capability registers, including a hardware-maintained tag bit. When an integer instruction writes on a register containing a capability, the tag is cleared automatically, preventing capability forgery via integer operations.
- The **data bus** is extended to **33 bits**: 32-bit data is extended by a tag bit to distinguish capabilities from integer data stored in the most significant bit (MSB).

- The **load-store unit** is modified to support atomic capability load/store (clc, csc) with tag checking and propagation
- **Special Capability Registers (SCRs)** are added (e.g. MTCC replacing mtvec and MEPCC replacing mepc) and the Program Counter Capability Register (PCC), which serve special purposes.

Pipeline architecture

At its foundation, the CHERIoT-Ibex pipeline may be instantiated as either a two-stage design (instruction fetch/decode (IF/ID) and execute/write-back (EX/WB)) or a more performance-oriented three-stage design (IF, ID, EX/WB)) [7]. The CHERIoT extensions integrate into this pipeline by augmenting the standard instruction path with capability-aware decode and execution units.

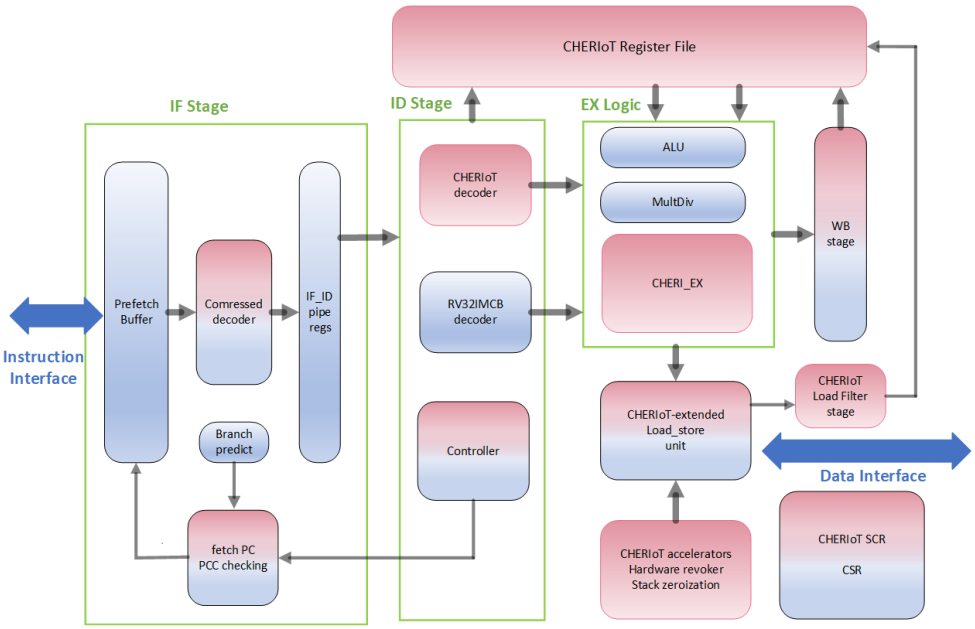


Figure 2.1: The CHERIoT-Ibex pipeline [7]

IF/ID Stage. As shown in Figure 2.1, the IF stage incorporates prefetch buffering, compressed instruction decoding, and program counter capability (PCC) checks, ensuring that all instruction fetches are capability-constrained. The ID stage then employs dual decoders: a standard RV32IMC decoder for base RISC-V instructions and a dedicated CHERIoT decoder for capability instructions, both feeding into a shared control path.

EX/WB Stage. The EX stage integrates traditional ALU and multiply/divide units with the CHERI_EX execution unit, which implements capability-specific arith-

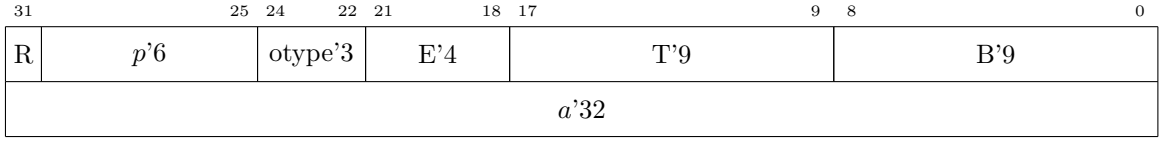
metic, bounds checking, and sealing/unsealing operations [6]. A CHERIoT-extended load/store unit mediates all memory accesses, enforcing capability permissions before issuing requests on the 33-bit data bus — an interface requiring two cycles to load or store a full 64-bit capability [7]. To support fine-grained memory safety, this unit interfaces with hardware accelerators, including the TBRE (The Background Revocation Engine) and STKZ (Stack Zeroization Unit), which operate in the background to enforce temporal and spatial safety guarantees. The write-back stage updates both integer and capability registers, with the CHERIoT load filter stage enforcing final checks on loaded data before it enters the register file.

Hazards. In CHERIoT-Ibex, data hazards are largely mitigated through operand forwarding. Control hazards occur on branches and interrupts, as the core has no branch prediction; taken branches and exception entries flush the pipeline and restart at the resolved target (calculated early in the pipeline), with PCC checks ensuring valid control flow. Memory-related hazards are more significant: capability loads and stores require multiple cycles over the 33-bit bus, and the load/store unit is a shared resource between the CPU and tightly coupled accelerators (TBRE and STKZ), so contention can introduce stalls when both issue memory requests.

Capability encoding

Capability size. A central challenge in CHERIoT is fitting capabilities into a 32-bit format. Standard CHERI capabilities are 128 bits; CHERIoT compresses them to suit embedded contexts. The end result is a compression format that fits all capability metadata into one 32-bit word, which — together with the 32-bit address — forms a 64-bit capability. The extended 33-bit data path can thus load a capability in 2 words (metadata + address), where each half carries a validity bit in the MSB tag bit. The two tags are then ANDed to ensure validity across both words; effectively ensuring that the entire capability is considered valid only when both words are tagged valid. The concrete capability format as described in [25] is shown in Figure 2.2; permission, bounds and sealing type compressions are explained in more detail below.

Permissions encoding. CHERIoT represents the 12 architectural permissions in a compressed 6-bit encoding by eliminating invalid or redundant combinations. For example, in CHERIoT, certain permissions are mutually exclusive (*execute* and *write* cannot coexist), while others are only meaningful together (*system register access* implies *execute* permission) [6, 25]. This encoding achieves a compact representation while still covering the full set of meaningful authority patterns.



R a reserved bit; normally zero, but may be set for untagged data (details omitted for brevity)
p a 6-bit compressed permissions field
otype a 3-bit 'object type' used for sealing capabilities
E a 4-bit exponent used for the bounds encoding
T a 9-bit top used in the bounds encoding
B a 9-bit base used for the bounds encoding
a the 32-bit address of the capability

Figure 2.2: Capability format as described in [25]

As a consequence of this compression system, no capability can possess all 12 architectural permissions. Instead, there are three *capability roots*: the write root, the executable root and the sealing root [25]. Root capabilities with those permissions are used to derive all further capabilities needed.

Bounds compression. The two 32-bit capability bounds (*base* and *top*) are stored in a compressed format relative to the address using only 22-bits for both bounds. This compression is achieved by 'sharing' some bits with the address, imposing certain restrictions on bounds-address combinations that can be expressed in this format. The bounds are represented using two 9-bit fields for mid-base and mid-top bits (`B` and `T`) and a 4-bit exponent field (`E`) taking values 0-14 or the special value 24 (encoded as 1111) to represent a capability spanning the entire address-space. The middle bound bits `B` and `T` are then inserted in the middle bits of the 32-bit address `a` as shown in Table 2.1.

address, $a =$	$a_{\text{top}} = a[31 : e+9]$	$a_{\text{mid}} = a[e+8 : e]$	$a_{\text{low}} = a[e-1 : 0]$
base, $b =$	$a_{\text{top}} + c_{\text{b}}$	B	0
top, $t =$	$a_{\text{top}} + c_{\text{t}}$	T	0

Table 2.1: Decoding formula for the bounds as shown in [25]

As part of this, the top bits of the bounds are 'corrected' to ensure that the bounds remain the same for addresses in the so-called *representable range* $[b, b + 2^{e+9})$. The corrections c_b and c_t are calculated according to the following formulae [25]:

$$c_b = \begin{cases} -1 & \text{if } a_{mid} < B \\ 0 & \text{otherwise} \end{cases} \qquad c_t = \begin{cases} -1 & \text{if } a_{mid} < B \text{ and } T \geq B \\ 1 & \text{if } a_{mid} \geq B \text{ and } T < B \\ 0 & \text{otherwise} \end{cases}$$

As described above, this encoding supports only a restricted range of bounds-address combinations, referred to as *representable*. Any modification of a capability's address or bounds therefore requires a *representability check* to confirm that the new values can be expressed in the encoding. If the check fails, the hardware either adjusts to the nearest representable values or clears the validity tag of the capability.

Sealing types. The architectural *otype* field is reduced to 3 bits in CHERIoT, yielding two disjoint sets of seven values (0 being reserved for unsealed capabilities) distinguished by the *execute* permission of the capability. This is done as in practice, software typically uses different types for executable and data capabilities. Five of the executable *otypes* are consumed by hardware-reserved sentry capabilities, leaving two for software-defined use, while the data *otypes* are left entirely to the RTOS, which reserves four for core components and exposes three to user code. Although this represents a stricter limit than CHERI's general sealing model, the CHERIoT RTOS layers a virtualized sealing abstraction over hardware support, providing sufficient flexibility for compartmentalization and object encapsulation in practice. [6].

2.2 CHERIoT RTOS

The CHERIoT RTOS represents a clean-slate operating system design specifically architected to exploit the capability-based security features of the CHERIoT hardware platform with the principles of least privilege and intentionality at its center [24]. Unlike traditional embedded operating systems that rely on memory management units for isolation, the CHERIoT RTOS implements a compartmentalization model based entirely on CHERI capabilities, enabling fine-grained privilege separation with minimal performance overhead [26].

Core Architecture and Organization. The CHERIoT RTOS is organized around a set of privilege-separated components that form the trusted computing base (TCB). The architecture eliminates the concept of an omnipotent kernel, instead distributing system functionality across multiple isolated components with minimal necessary privileges [26].

The core components include:

- **Switcher**: The most privileged component responsible for context switching between threads
- **Allocator**: Manages heap allocation with compartment-aware memory safety and exposes an API for sealing

- **Scheduler**: Implements thread scheduling policies while maintaining compartment and thread isolation
- **Loader**: Runs when a firmware image is loaded into memory; receives the root capabilities from hardware and provides restricted capabilities to compartments and threads

Compartmentalization. Each component operates within its own protection domain, with inter-component communication occurring exclusively through well-defined capability-based interfaces. This architectural approach ensures that compromise of any single component cannot escalate to full system control; this means that security properties can be verified locally within individual components, rather than requiring system-wide invariant maintenance.

Unsealer

The unsealer component, implemented primarily in `token_unseal.S` within the token library, provides the fundamental service of converting sealed capabilities back to their unsealed, usable form. It is architecturally integrated as part of the memory allocator component, which exposes a sealing API; the library supports both dynamic sealing (runtime seal / unseal operations) and static sealing (predefined, compile-time sealed capabilities), which is why it provides two permit-unseal capabilities. This addresses a critical limitation of the CHERIoT platform: the hardware sealing type field is only 3 bits, providing space for just 7 different sealing types, while a system could easily have more than 7 compartments requiring sealing services and/or require multiple sealing types per compartment [26].

Security Role. In the RTOS compartmentalization model, the unsealer serves as a critical trust boundary component. It enables secure capability delegation patterns where sealed capabilities can traverse untrusted compartments without exposing their underlying authorities until properly authenticated unsealing occurs [11]. Each sealed capability includes a header that encodes its type and other metadata. Leakage of this header would be problematic because it serves as the type identifier; if an untrusted component could modify it, they could change the type of a sealed capability, subverting the type-safe guarantees that sealing provides.

The allocator (and the loader, for static sealed objects) virtualizes the sealing abstraction through several key mechanisms:

- **Sealing Capability Generation**. Callers can request new capabilities that work exclusively with the allocator's sealing API, bypassing the direct hardware `cseal` / `cunseal` instruction usage.
- **Dual-Capability Allocation**. Memory can be allocated via an API that returns both a sealed and an unsealed capability to the same memory object, so that trusted code can use the memory immediately while a sealed capability is available for delegation to untrusted compartments.
- **Header-Base Authentication**. Sealed capabilities include a header containing authentication information for subsequent unsealing operations.

Technical Implementation. The unsealing process is provided by the user with the sealed object pointer and the user's sealing key, which authorizes the specific type of the sealed value. In addition, the procedure relies on a global unsealing authority private to the token library — the hardware unseal key — which can unseal any sealed capability. The unsealing operation proceeds in a series of checks and transformations that enforce both authority and type safety:

1. **Authority Validation**: Verifying that the caller's sealing key authorizes unsealing (e.g., possesses `PERMIT_UNSEAL` permissions)
2. **Hardware Unseal Check**: Use the library-private hardware unseal key to convert the sealed token into a valid unsealed capability (`CUNSEAL` instruction)
3. **Type Matching**: Ensure that the sealed capability's object type matches the sealing key's permitted types (software check via `BNEQ`)
4. **Capability Reconstruction**: Adjust the bounds and offset of the capability to produce a usable unsealed capability, effectively removing the sealed header

A critical aspect of the unsealing implementation is the requirement to 'cut off the header' of the sealed data during the unsealing process before returning the unsealed capability. This header removal is necessary because sealed objects contain metadata that must not be exposed to the receiving compartment but is essential for the unsealing operation's integrity verification.

The `token_unseal.S` implementation uses a specific register allocation strategy to maintain the ability to reason about security during the unsealing process:

- `ca0` : Contains the user's sealing key capability and is replaced by the unsealed object pointer upon successful unsealing

- `ca1`: Holds the sealed object capability to be unsealed
- `ca2`: Holds the unsealing authority and stores intermediate capabilities during processing
- `a3`: Used for computational operations during the unsealing procedure

This register allocation creates security concerns because sensitive data — such as the unsealing authority and intermediate states of the unsealed object pointer including the header — temporarily reside in processor registers and must be ensured not to be accessible to the user's code. We now turn to the formulation of the security properties that should hold in the unsealer.

Security properties. The most fundamental security property of the unsealer is that the unsealing authority must never remain in the register file after the unsealer's execution. The unsealing authority is a powerful authority that could enable the user to unseal other code's sealed capabilities and thus compromise the entire sealing-based security model if exposed. Key protection requirements include:

- **Register Isolation**: Ensuring that the register `ca2` holding the unsealing authority is correctly clobbered both upon success and failure
- **Authority Containment**: Preventing the unsealing authority from being duplicated or otherwise modified and exposed

The second critical property ensures that the sealed object pointer is only exposed after full completion of the successful unsealing operation — premature exposure of the unsealed capability could expose the sensitive metadata stored in the header. This property encompasses primarily the *capability integrity*: guaranteeing that the unsealed object pointer exposed to the user has the correct bounds, excluding the header.

Formal verification of these properties is highly desirable to guarantee that the complex register manipulation sequences maintain security invariants throughout the execution. Special care is to be given to the validation of the security properties even on failure paths and especially at exception traps.

Switcher

Core functionality. The switcher is the most privileged component of the RTOS — running with access to untrusted data — and serves as the central mechanism for context switching between both compartments and threads within the CHERIoT

RTOS. According to the CHERIoT RTOS documentation [26], the switcher's responsibilities include saving register state on interrupts, invoking the scheduler with sealed references to the register save area and trusted stack, unsealing and setting up the target context on cross-compartment calls (while shrinking and clearing the stack as required), and reversing these operations on return.

Common context installation. After saving the outgoing state and selecting the next runnable context, the switcher invokes the common context install procedure (`.Lcommon_context_install` in `entry.S`). This procedure restores the selected context's saved state from the *TrustedStack*, including general-purpose and capability registers, the stack pointer and the program counter. Once these are installed, execution resumes from the restored program counter. The install procedure thus complements the save procedure, together forming the core of CHERIoT's context switching mechanism.

Security. From a security perspective, the common context installation procedure presents several critical verification challenges. Since the switcher executes with 'Access System Registers' (ASR) permission, it is essential to verify that the newly installed program counter capability (PCC) never retains ASR permissions, thereby preventing privilege escalation into system-level authority. Equally important is the assurance that all registers, and in particular the capability stack pointer (CSP), are fully sanitized to eliminate residual state that could enable cross-context information leakage. These properties to be proven by formal verification underpin the integrity and isolation guarantees of the switcher.

Chapter 3

Formulating Safety Conditions

In this chapter, we develop a rigorous framework for reasoning about safety properties of capabilities in the CHERIoT-Ibex system. We first introduce a precise mathematical model of capabilities, capturing their metadata, bounds, permissions, and object types. Building on this formalization, we define notions such as representability and derivability, which allows us to express the safety conditions that must hold at the end of code execution.

3.1 Mathematical representation of capabilities

To begin reasoning about safety conditions regarding capabilities, we need a precise mathematical model of their structure. As described in Section 1.2, representations of capabilities typically include 5 fields of metadata: the address, the bounds, the permissions, the object type and the validity bit. We can replicate this mathematically, as is done in Definition 1, applied to the (uncompressed) representation used in the CHERIoT-Ibex. This definition is the basis for further discussion of capability authority, its restriction, transfer and what we want to allow or forbid in a system.

Definition 1 (Capability) *A* capability *is a 5-tuple:*

$$c = (tag,\ a,\ (b, t),\ perms,\ otype)$$

where

- $tag \in \{0, 1\}$ *is the validity bit,*
- $a \in \mathbb{A}_{32}$ *is the address,*

- $(b, t) \in \mathbb{A}_{32} \times \mathbb{A}_{32}^*$ *are the lower and upper bounds,*
- $perms \subseteq \mathcal{P}$ *is the set of permissions, and*
- $otype \in \mathcal{T} \cup \{\bot\}$ *is the object type, with* $\bot$ *denoting an unsealed capability.*

Here, $\mathbb{A}_{32} = \{0 \,..\, 2^{32} - 1\}$ *is the address space,* $\mathbb{A}_{32}^* = \mathbb{A}_{32} \cup \{2^{32}\}$ *is the extended address space,* $\mathcal{P}$ *denotes the set of architectural permissions, and* $\mathcal{T}$ *is the finite set of sealing types.*

In the CHERIoT-Ibex, the set of permissions consists of 12 elements (i.e. $|\mathcal{P}| = 12$), which are specified in [25], while the set of object types is $\mathcal{T} = \{1 \,..\, 7\}$, which — as explained in Section 2.1 — decodes to 7 sealing types each for *execute* and *write* capabilities.

Representability

A critical aspect of the CHERIoT-Ibex implementation of capabilities is the compressed encoding format (see Section 2.1). Each capability must fit into two 32-bit words so that it can be efficiently loaded or stored in just two cycles. This micro-architectural constraint is achieved via two mechanisms:

- The hardware uses a **4-bit shared exponent** for both bounds, along with one 9-bit field for each bound. This design restricts the combinations of lower and upper bounds that can be represented, motivating Definition 2.
- The bounds share the top bits with the **address field**, further reducing space. For correct and consistent decoding, the address and bounds must lie in a compatible range. In practice, this is checked by asserting that the address is in the so-called **representable range** relative the bounds (Definition 3).

Understanding which bounds and addresses are representable is essential because operations that change bounds or addresses — for example when deriving new capabilities — must ensure that the requested values can actually be expressed in hardware. In CHERIoT-Ibex such representability checks are performed directly in the microarchitecture, and our verification does not remodel them explicitly. Nevertheless, they contribute to the overall complexity of reasoning about capability behaviour. The notion of *representability* captures exactly which bounds can be stored and which addresses are valid for those bounds.

Definition 2 (Representable bounds) *A tuple of bounds* $(b, t) \in \mathbb{A}_{32} \times \mathbb{A}_{32}^*$ *is said to be* representable *if either*

- $b = 0$ *and* $t = 2^{32}$ *(encoding the entire address space), or*
- *There exists an exponent* $e \in \{0 \,..\, 14\}$ *such that*

  1. $b, t \equiv 0 \pmod{2^e}$*, and*
  2. $0 \leq (t - b) \leq 2^e \cdot 511$.

We write $\mathcal{B}$ *for the subset of* $\mathbb{A}_{32} \times \mathbb{A}_{32}^*$ *of representable bounds.*

This definition encodes the constraints imposed by the CHERIoT-Ibex compression:

- The **exponent** e controls the granularity of alignment. A larger e means coarser alignment but allows larger ranges.
- The **length restriction** $t - b \leq 2^e \cdot 511$ ensures the range fits within the limited 9-bit fields for bounds, shifted by the exponent.
- The first case allows capabilities that cover the **entire address space**, which is represented by an exponent value 24.

Intuitively, representable bounds are all combinations of bounds that the hardware can encode using two 9-bit fields and a shared 4-bit exponent.

Definition 3 (Representable range) *Let* $(b, t) \in \mathcal{B}$*. The* representable range $\mathcal{R}(b, t)$ *is defined as*

$$\mathcal{R}(b,t) = \begin{cases} \mathbb{A}_{32}, & \text{if } b = 0 \text{ and } t = 2^{32} \\ [b, b + 2^{e+9}), & \text{otherwise} \end{cases}$$

where $e \in \{0 \,..\, 14\}$ *is the maximal exponent witnessing representability of* (b, t)*.*

We say that (b, t) *is* representable with respect to an address $a \in \mathbb{A}_{32}$ *if* $(b, t) \in \mathcal{B}$ *and* $a \in \mathcal{R}(b, t)$*.*

Even if bounds are representable, the capability's current pointer must lie within a segment that the hardware can reconstruct consistently. The representable range captures this requirement: it defines the set of addresses for which the bounds can be decoded and decoded correctly. The range $[b, b + 2^{e+9})$ arises because the top bits of the address are shared with the bounds. Only the first $e + 9$ bits may change without altering the top bits, yielding a length of 2^{e+9}.

Equivalence with hardware encoding. The notion of *representability with respect to an address* captures exactly the conditions under which a bounds-address pair can be encoded in the CHERIoT-Ibex. This equivalence is formalized in theorem 1, which guarantees that our mathematical definitions of representable bounds and ranges align precisely with the hardware encoding. This allows rigorous reasoning about which capabilities can exist in the system.

Theorem 1 *A pair $(b, t) \in \mathbb{A}_{32} \times \mathbb{A}_{32}^{*}$ is representable with respect to an address $a \in \mathbb{A}_{32}$ if and only if there are $B, T \in \mathbb{A}_9$ and $E \in \mathbb{A}_4$ such that (b, t) are the decoded base and top bounds according to the procedure of Section 2.1 when executed with B, T, E and address a.*

The theorem can be proven using standard discrete mathematics reasoning over the encoding procedure.

Derived capabilities

The CHERI architecture provides a fixed set of operations for transforming capabilities (see Section 1.2). These operations are monotonic in the sense that they cannot create new authority: any capability produced from another has bounds no larger, permissions no greater, and cannot regain validity once invalidated. From a security perspective, this monotonicity is crucial: if a sensitive capability is not to be leaked, then it is insufficient to check that the exact capability does not appear in registers or memory. One must also ensure that no *derived capability* — any authority that originates from the sensitive capability — is present, since even restricted forms of authority may enable unintended behaviour.

Formally, derivability captures the idea that c' can be obtained from c via a sequence of valid CHERIoT capability operations. It is mathematically formulated

in Definition 4 and its implementation in SystemVerilogAssertions (SVA) is shown in Figure 1 in the appendix.

Definition 4 (Derivable Capabilities) *Let c, c' be capabilities. We say that c' is* derivable *from c and write $c' \preceq c$ if:*

1. *$tag(c) = 1$: only valid capabilities can be the basis for derivation,*
2. *$[b(c'), t(c')) \subseteq [b(c), t(c))$: the derived capability c' has bounds at least as strict as the source capability c, and*
3. *$perms(c') \subseteq perms(c)$: the derived capability does not possess any permissions that the source capability does not.*

Thus, c' represents a restriction of c: it cannot restore validity, can access no more memory and enables no more operations. The derivability relation $\preceq$ has the following structural properties, following from the structural properties of $\subseteq$.

- **Reflexivity**: $c \preceq c$ for any valid capability c.
- **Transitivity**: If $c' \preceq c$ and $c'' \preceq c'$, then $c'' \preceq c$.

Hence, $\preceq$ defines a partial order on the set of valid capabilities. Intuitively, this order captures containment of authority: higher elements confer as least as much authority as those below them.

3.2 Safety conditions

In this section, our aim is to formulate the safety conditions we want to hold at the end of a code sequence. These conditions are naturally phrased as postconditions: they describe the state that must be achieved once execution completes. Naturally, our ultimate goal is more complex: we want to prove these conditions hold whenever the code is executed. That broader question will be taken up later (see Chapter 4). Here we deliberately restrict our focus to the conditions themselves, as precise end-of-execution invariants.

Ensuring non-derivability in registers

Typically, we are interested in safety properties of the form 'the at-entry value of `c` or any capabilities derivable from it, must not be present anywhere in the register file upon completion of the code sequence.'

Here, `c` denotes a variable holding a sensitive capability. To express this property precisely, we must clarify four aspects:

1. The meaning of the **at-entry value** of `c`,
2. What it means for a capability to be **derivable**,
3. What it means for a capability to be **absent from the register file**,
4. When a **code sequence has completed**.

Items 1 and 4 are closely tied to questions of pipeline state and execution timing and will therefore be addressed in Chapter 4. For the present discussion we assume that variables are available which refer to the at-entry value of the relevant capabilities, and focus on the remaining two aspects: derivability (2) and register file contents (3).

```
function automatic bit no_derivatives_except_ca0_ca1_fn(reg_cap_t cap, logic [31:0] addr);
    for (int i =0; i <32; i++) begin
        if (i!=10 &&i!=11) begin
            if(derived_from(`RF.rf_cap[i], regs[i], cap, addr))
                return 0;
        end
    end
    return 1;
endfunction
```

Figure 3.1: Implementation of register file scanning for derivable capabilities in SVA

Implementation. Derivability is defined formally in Definition 4. In our SVA implementation (see Figure 1 in the appendix), this check is realized by decompressing the capabilities and comparing their permission sets and bounds. Although checking directly on the compressed representation would be possible, decompression is preferred for clarity, with little cost in performance. To enforce the register file condition, we provide a function (see Figure 3.1) that takes the sensitive capability `c` as input and iterates through all 32 capability registers. Each register is checked for derivability from `c`. If a derivable capability is found, the function terminates immediately and returns `False`. If the iteration completes without finding any derivable capability, the function returns `True`, indicating that the register file is safe. Optional exclusions (e.g. the register holding `c` itself) can be specified to avoid trivial

self-derivation, as is done for registers `ca0` and `ca1` in the example function shown in Figure 3.1.

Formally, this function can be defined as the following

$$not\_derivable(cap, S \subseteq \{1 \mathinner{..} 32\}) \iff \bigwedge_{i \in S} (c_i \not\preceq cap)$$

where *cap* is the sensitive capability and c_i is the capability in register i and S is the set of indeces to consider.

Unsealer safety conditions

We now apply these concepts to formulate the safety conditions for the unsealer module. At the entry point of the unsealer, two sensitive capability objects are available:

- the sealed object pointer `obj_ptr`, and
- the global unsealing authority `us_auth` private to the token library (see Section 2.2).

At completion of the unsealer's execution:

- the **unsealing authority**, or any of its derivatives, must not appear in the register file, and
- the **object pointer** may only appear in two specific registers:
  - in register `ca1` in its original sealed form,
  - in register `ca0` in its correctly unsealed form (i.e. with the header removed).

Everywhere else in the register file, neither `obj_ptr` nor `us_auth` (nor any derived capabilities) may be present. For convenience, we assume that the variables `obj_ptr` and `us_auth` represent the at-entry values of the sealed object pointer and unsealing authority respecively, and by slight abuse of notation we treat them as including their addresses. The formal safety conditions are:

1. `obj_ptr` and all derivable capabilities must not be present in the register file.
2. `us_auth` and all derivable capabilities must not be present in the register file.
3. Exceptions:
   (a) Register `ca0` may hold the correctly unsealed object pointer.

```
// allow ca1 to be exactly the sealed pointer
logic ca1_ok =~ca1.valid ||(ca1 ==obj_ptr &&a1 ==obj_ptr_addr);
// allow ca0 to hold the unsealed pointer with correct bounds
logic ca0_ok =~ca0.valid ||is_correct_unsealed_pointer(ca0, a0);

// check if the unsealing authority is safe (no derivatives anywhere except in the special allowed
    ↪ cases)
logic us_auth_safe_other_regs =no_derivatives_except_ca0_ca1_fn(us_auth,us_auth_addr);
logic us_auth_safe_ca1 =ca1_ok ||~derived_from(ca1, a1, us_auth,us_auth_addr);
logic us_auth_safe_ca0 =ca0_ok ||~derived_from(ca0, a0, us_auth,us_auth_addr);
logic us_auth_safe =us_auth_safe_other_regs &&us_auth_safe_ca1 &&us_auth_safe_ca0;

// check if the object pointer is safe (no derivatives anywhere except in the special allowed cases)
logic obj_ptr_safe_other_regs =no_derivatives_except_ca0_ca1_fn(obj_ptr, obj_ptr_addr);
logic obj_ptr_safe_ca1 =ca1_ok ||~derived_from(ca1, a1, obj_ptr, obj_ptr_addr);
logic obj_ptr_safe_ca0 =ca0_ok ||~derived_from(ca0, a0, obj_ptr, obj_ptr_addr);
logic obj_ptr_safe =obj_ptr_safe_other_regs &&obj_ptr_safe_ca1 &&obj_ptr_safe_ca0;
```

Figure 3.2: SVA implementation of the final safety conditions for the unsealer module

(b) Register `ca1` may hold the sealed object pointer in its original form (`obj_ptr`).

Implementation. In the SVA implementation (Figure 3.2), we encode the two exceptions explicitly via the signals `ca0_ok` and `ca1_ok`. Invalid capabilities are ignored, since they cannot be revalidated. For each sensitive capability, we then check:

- whether one of the exceptions applies, and
- if not, whether `ca0` / `ca1` do not hold a derivable capability.

Finally, the register file scanning function described above is used to verify that no other register in the file holds a capability derivable from `obj_ptr` or `us_auth`. All results are combined into the predicates `us_auth_safe` and `obj_ptr_safe`, representing the final safety conditions of the unsealer. The implementation is shown in Figure 3.2.

Switcher safety conditions

The complete verification of the switcher is out of scope for this thesis. However, to illustrate how our methodology generalizes beyond the unsealer, we also consider a fragment of the switcher's *common context install* procedure (see Section 2.2). the details of this setup, including the simplifying assumptions required to make it tractable, are discussed later in Chapter 5. Here we state only the safety conditions that guide the verification.

Safety conditions. At entry, the only sensitive value is the capability stack pointer `csp`, which is used to restore the processor state and then quarantined in

`mtdc`. It must not be leaked elsewhere in the register file. Let `csp_at_entry` denote the at-entry value in the register `csp`. At completion of the install block (jsut before `mret`), the following conditions must hold:

1. **Persistence of** `csp_at_entry`: The at-entry value of `csp` is successfully installed in the system register `mtdc`.
2. **Non-leakage of** `csp_at_entry`: No register in the capability register file may contain a capability derivable from `csp_at_entry`.

Together, these conditions express that the at-entry value of `csp` is **quarantined** in `mtdc` and otherwise eliminated from the architectural state. Formally, this can be captured in the formula

$$\bigwedge_{i=1}^{32} (c_i \not\preceq csp_0) \wedge mtdc = csp_o$$

where c_i is the capability present in capability register i and csp_o stands for the capability `csp_at_entry`.

Chapter 4

Logical Encoding of Code Execution

As explained in Chapter 3, we are not just interested in proving isolated state invariants; we need to show that these invariants hold *throughout real code execution*. This requires a way to express, in logic, that a sequence of instructions is actually executed on the processor.

Approach. In this chapter, we develop such a methodology for reasoning about binary code execution on the formal model of the CHERIoT-Ibex processor. Our approach is based on observing instructions at the writeback (WB) stage of the pipeline, where execution is guaranteed to have completed. Using a small set of indicator signals, we can encode binary execution paths as temporal properties and extend this encoding to cover timing, exceptions, branching and the capture of 'at-entry' values needed for safety properties.

4.1 Observation strategy: the writeback stage

Motivation

When reasoning about binary execution on a pipelined processor, there are two natural places to observe instructions:

- **At entry (IF/ID stage)**, where instructions are fetched and prepared for execution.
- **At writeback (WB stage)**, where results are committed and execution is guaranteed to have completed.

At entry. At first glance, the IF/ID stage seems appealing: it captures instructions at the moment they enter the pipeline. However, instructions observed at entry may never execute: branches can flush them, exceptions can interrupt them and stalls make it unclear *when* they might complete. Encoding execution at entry would therefore require reasoning about hazards, timing and dependencies — all of which quickly complicate verification.

At writeback. The writeback stage, by contrast, provides what we call the *point of truth*. An instruction reaching WB has traversed all earlier pipeline stages and is about to update the architectural state. From the perspective of verification, this provides us with a clean and unambiguous marker that the instruction has indeed executed; hazards and stalls are resolved before WB. The price of this simplicity is that we shift certain responsibilities elsewhere: we implicitly assume that earlier stages behave correctly (e.g. no interrupts), and we treat branches and exceptions separately.

Indicator signals and assumptions

Assumptions. We adopt several high-level assumptions to simplify reasoning about timing and control; these have been reviewed with SCI Semiconductors, who confirmed they are fair in practice. .

- **Instructions run at minimum latency**: The latency depends on the instruction type: most are single-cycle, while (capability-) loads and stores may take two or three cycles.
- **Memory returns within one cycle**: Together with the previous assumption, this eliminates combinatorial timing blow-ups. For example, two instructions each taking between one and three cycles already create nine possible timing combinations. Fixing latency keeps the verification state space manageable.
- **No interrupts during execution**: For the sequences we check, interrupts are disabled by context, so we may safely assume they do not occur.

Indicator signals. To make the WB stage usable as a logical observation point, we rely on a small set of *indicator signals* introduced in the follower module by Ploix [27]. These are not part of the RTL design itself but are additional instrumentation for formal reasoning about CHERIoT-Ibex. In model checking, the RTL and the follower are combined so that properties can be expressed over both hardware state

and these derived indicators. The full implementation of the follower is provided in Figure 3 in the appendix.

Exception handling. The writeback stage in CHERIoT-Ibex also serves as the point where exceptions are latched. When an instruction raises an exception in EX, its decoded bits and status are still passed into WB, where they remain until the exception is fully handled. The prefix `wbexc_*` in the indicatoor signals reflects this dual role: they describe the instruction resident in WB, regardless of whether it is committing normally or waiting due to an exception.

Latching. The follower latches values into the `wbexc_*` registers whenever the signal `instr_will_progress` is true. At that hand-off, the instruction in EX moves into WB, and the follower records the decompressed instruction bits, the branch decision and any associated fetch/exception flags. If WB later finishes, `wbexc_exists` is cleared, creating a bubble or letting the executing instruction proceed to WB.

We use the following signals as our vocabulary for describing execution:

- `wbexc_decompressed_instr` : the 32-bit (decompressed) instruction latched into WB. Ignore its value when `wbexc_exists` is false (bubble). Is `wbexc_err` is true, this still holds the instruction that raised the exception; the handling of the exception is modelled separately.
- `instr_will_progress` : signals that the pipeline will advance to the next cycle, i.e. the instruction currently in EX will move into WB. In principle this can coincide with exceptions, but in our usage it marks non-exceptional progression: if an exception had occurred, handling would take more than one cycle and `instr_will_progress` would not line up with the next WB instruction.
- `wbexc_exists` : true whenever the WB stage holds a valid instruction (or an exception). If false, WB is a bubble and associated fields are meaningless.
- `wbexc_has_branched` : the branch-taken decision for the instruction that just advanced to WB. Only meaningful if that instruction is a branch instruction.
- `wbexc_fetch_err` : indicates that the instruction in WB encountered a fetch error.
- `wbexc_err` : true when the WB slot corresponds to an exception rather than a normal commit. This can result from an error detected in EX or from exception handling in WB.

These indicators give us exactly the hooks to say ‘instruction X is meaningfully in WB’, ‘the pipeline advances now’, and ‘a branch was taken here’, without exposing

low-level hazard and control logic. In the next section, we use them to encode concrete sequences of binary instructions as temporal properties.

4.2 Encoding instruction sequences

Base encoding

To reason about execution, we first introduce a macro that tells us when a specific binary instruction is present in the WB/exception stage:

```
1 `define INSTR wbexc_decompressed_instr
2 `define INSTR_WB(instr) \
3    wbexc_exists &&~wbexc_fetch_err &&`INSTR ==instr
```

Intuitively, `INSTR_WB(i)` is true whenever the decompressed instruction in WB matches `i`, WB is not a bubble, and no fetch error occurred. In other words: 'the instruction `i` has been correctly fetched and has now reached WB/exception handling'. Whether an exception is/was raised will be handled separately in Section 4.3.

Logical perspective (LTL). The goal is to describe *traces through WB*, where one instruction appears, completes, and the next one follows. For clarity, we use shorthand state predicates when writing temporal logic:

- $wb(l)$: instruction l is in WB (from `INSTR_WB(l)`)
- *ready*: the pipeline will progress (from `instr_will_progress`)

With this notation, a simple step looks like:

$$(wb(l_0) \wedge ready) \wedge \mathbf{X} wb(l_1) \tag{4.1}$$

which reads: 'the instruction labelled l_0 is in WB, the pipeline is ready progress and in the next cycle l_1 is in WB'. If the instruction l_0 takes more than one cycle (e.g. a load), we capture that explicitly:

$$(wb(l_0) \wedge \neg ready) \wedge \mathbf{X} ready \wedge \mathbf{X}\mathbf{X} wb(l_1) \tag{4.2}$$

This says 'l_0 is in WB but does not progress, one cycle later the pipeline is ready to progress, and one cycle after that l_1 appears in WB'. By construction, $\neg ready$ implies that the instruction in WB does not change, so $wb(l_0)$ does not need to be restated in the stalled cycle. From these patterns,longer code sequences are built by chaining such formulas, ensuring that instructions appear in WB exactly in order, with explicit timing and no bubbles or spurious instructions in between.

Encoding in SVA. In practice, we express these properties in SystemVerilogAssertions (SVA). Here, the operator `##1` plays the role of the LTL 'next' operator **X**. The operator stacks naturally:

- `a ##1 b` $\equiv a \wedge \mathbf{X}b$
- `a ##1 b ##1 c` $\equiv a \wedge \mathbf{X}b \wedge \mathbf{XX}c$
- in general, every element after a `##1` is shifted one more cycle into the future

For example, the stalled case above (4.2) translates directly to:

```
sequence stalled_instruction;
    (`INSTR_WB(l0) &&~instr_will_progress
    ##1 instr_will_progress
    ##1 `INSTR_WB(l1));
endsequence;
```

Here, the first line says 'l_0 is in Wb but does not progress', the second line says 'one cycle later the pipeline is ready', and the third line says 'one more cycle later l_1 is in WB'. To illustrate how binary code is translated into our WB-based encoding, consider the following fragment from the unsealer binary:

```
0: fe4506db  ct.cgettag a3, ca0 ; l00_cgettag
4: c281  beqz a3, .Lexit_failure ; l04_beqz
6: fe2506db  ct.cgetbase a3, ca0 ; l06_cgetbase
a: 00d51063  bne a0, a3, .Lexit_failure ; l0a_bne
e: fe3506db  ct.cgetlen a3, ca0 ; l0e_cgetlen
```

This sequence of instructions corresponds, in our base encoding, to the following SVA fragment:

```
sequence straight_line_example;
    (`INSTR_WB(l00_cgettag) &&instr_will_progress
    ##1 `INSTR_WB(l04_beqz) &&instr_will_progress
    ##1 `INSTR_WB(l06_cgetbase) &&instr_will_progress
    ##1 `INSTR_WB(l0a_bne) &&instr_will_progress
    ##1 `INSTR_WB(l0e_cgetlen) &&instr_will_progress);
endsequence;
```

The structure of the encoding is straightforward: each instruction from the binary is represented as an `ISNTR_WB(l)` predicate in WB, conjoined with the requirement that the pipeline progresses, and linked to the next instruction by `##1`. At this stage we ignore branching; the encoding will be refined in Section 4.2 to account for branch decisions.

Branching

So far we have considered straight-line execution: each instruction reaches WB, the pipeline progresses, and the next instruction follows in order. Real binaries, however, often contain *branch instructions*, which may either continue along the fall-through

path or or redirect control flow to a branch target. At the WB stage, this decision is visible through the indicator signal `wbexc_has_branched` .

The signal `wbexc_has_branched` records the branch decision made in EX for the instruction that has just entered WB. When this flag is true, the pipeline does not also assert `instr_will_progress` in the same cycle. Intuitively this makes sense: a taken branch flushes the pipeline, so progression to the next sequential instruction does not occur. Instead, the pipeline produces a one-cycle bubble in WB before the branch target arrives, since CHERIoT-Ibex (without branch prediction) already prefetches and decodes the branch target once the branch is identified (see Section 2.1.

Logical perspective (LTL). We extend the shorthand predicates from Section 4.2 with two additional symbols:

- *br*: the current WB instruction branched (from `wbexc_has_branched`)
- *bubble*: WB is empty (from `~wbexc_exists`)

Together with the semantic invariant discussed above ($br \wedge wb(l) \rightarrow \neg ready$), we can capture both outcomes of a branch. Let l_{br} denote the branch instruction, l_{next} its sequential successor, and l_{target} the branch target.

- **Fall-through (branch not taken):**

$$(wb(l_{br}) \wedge ready) \wedge \mathbf{X} wb(l_{next}) \tag{4.3}$$

- **Branch taken:**

$$(wb(l_{br}) \wedge br) \wedge \mathbf{X} bubble \wedge \mathbf{XX} wb(l_{target}) \tag{4.4}$$

Intuitively, at a branch instruction in WB, we can describe the outcome as:

- **Not taken**: use *ready* and step to l_{next}
- **Taken**: replace *ready* with *br*, insert one bubble, then continue at l_{target}

Practicality. In practice, we must explicitly encode the code sequence that follows *both* possibilities. Each outcome is expressed as its own sequence (e.g. success vs. failure), and properties are checked separately for each. This can in principle lead to exponential blow-up when branches are nested. In our case study, however, each branch instruction presents an 'off-ramp' from a short success path: execution either continues along the success path or jumps to a failure path of fixed length with no further branching. This containment keeps the encodings tractable.

Encoding in SVA. The following worked example extends the straight-line fragment used in Section 4.2 (see Code Snippet 4.2) with the branch at `l0a_bne` being taken. We keep `~wbexc_has_branched` explicit on the fall-through line for readability (even though `instr_will_progress` already implies it).

```
sequence branch_2_l06_cgetbase;
    (`INSTR_WB(l00_cgettag) &&instr_will_progress
    ##1 `INSTR_WB(l04_beqz) &&instr_will_progress &&~wbexc_has_branched
    // Branch at l04_beqz is not taken; execution continues normally
    ##1 `INSTR_WB(l06_cgetbase) &&instr_will_progress
    ##1 `INSTR_WB(l0a_bne) &&wbexc_has_branched
    // Branch taken at l0a_bne: use br, bubble, then jump to target sequence
    ##1 ~wbexc_exists
    ##1 failure_sequence);
endsequence;
```

How to read this. The fragment proceeds straight-line up to `l06_cgetbase`. At `l0a_bne`, we consider the **taken** case: the property checks `wbexc_has_branched`, enforces the one-cycle WB bubble, then continues at `failure_sequence`, the separately defined branch target path. The **fall-through** case can be seen earlier at instruction `l04_beqz`, where execution continues as normal.

4.3 Proving safety properties

In Chapter 3 we defined safety conditions on processor states, loosely of the form 'the at-entry value of this capability is not exposed'. At that point we treated *at-entry values* as given, without yet explaining how they are captured in the execution model.

The actual properties we want to prove, however, are not state conditions in isolation, but properties of the form

$$sequence \rightarrow condition$$

meaning: 'if a given code sequence executes, then at the point of exit the corresponding safety condition holds'. In Section 4.2 we developed a methodology for encoding concrete execution paths of a binary on CHERIoT-Ibex, including both straight-line execution and branching. This gives us the 'left-hand side' of the implication above: we can formally capture *all expected execution sequences up to a normal return.*

To complete the picture, two elements remain:

1. **At-entry values**, which let us make precise the notion used in Chapter 3 and allow safety conditions to compare the final state against the initial state of execution.

2. **Exceptions as exit points**, since execution can also leave a module through exception handlers, and the safety conditions must hold in those cases as well.

The remainder of this section develops these two extensions, thereby closing the gap between abstract safety conditions and concrete verifiable properties over binary execution.

Capturing at-entry values

The encodings in Section 4.2 capture the control flow of instruction sequences, but they do not yet let us relate the final state of execution to its intial state. Safety conditions from Chapter 3, however, are defined precisely in terms of such comparisons: for example, requiring that a capability's entry value is not present in the register file at exit. To reason about these properties, we therefore need a way to refer to the *at-entry value* of relevant registers or signals throughout the verification of a sequence.

Challenges. Naive strategies such as using a circular buffer of past values or fixed-cycle delays from the beginning of the sequence, proved unwieldy in SystemVerilogAssertions: dynamic indexing is not well supported, and fixed delays break down in the presence of branching or variable-length sequences. **Solution**. These limitations led us to a more direct approach:

- When the **first instruction** of the sequence enters WB, we **snapshot** the relevant register or signal into a dedicated `_at_entry` variable.
- This value is then held constant for the rest of the execution, providing a stable reference for safety conditions.
- This approach works reliably as long as the first instruction of the sequence is unique (so we know when to take the snapshot), which holds in our case study.

Implementation. To illustrate, consider the capability register `ca2` holding the unsealing authority. We define a companion variable `ca2_at_entry` that snapshots the value of `ca2` as soon as the first instruction of the sequence appears in WB:

```
reg_cap_t ca2_at_entry =NULL_REG_CAP;
always_latch begin
    if (($past(instr_will_progress) &&`INSTR ==ENTRY_INSTR &&wbexc_exists) ||~rst_ni) begin
        ca2_at_entry =ca2;
    end
end
```

- The guard `$past(instr_will_progress)` ensures that we only latch when the pipeline has just progressed (`instr_will_progress` was true in the previ-

ous cycle). That is, we snapshot exactly when the entry instruction **first** arrives in WB.

- The test `'INSTR == ENTRY_INSTR` checks that this instruction is the designated first instruction of the sequence.
- The signal `wbexc_exists` ensures that WB is valid (not a bubble).
- The clause `~rst_ni` reinitializes the snapshot on reset.

Together, these conditions guarantee that the snapshot is taken exactly once, at the moment the entry instruction first appears in WB. The variable `ca2_at_entry` then remains constant for the remainder of the execution (the entry instruction does not reappear in WB within the same sequence) and can be used in the RHS of safety properties. In practice, we generate such latches for many variables using a simple macro (shown in Figure 2 in the appendix), so that at-entry values can be defined systematically without duplicating code.

This approach isolates the *at-entry state* once and for all, so conditions like

$$code\_sequence \rightarrow (cap \preceq cap\_at\_entry)$$

are well-defined and meaningful. With at-entry values captured, we can now apply the safety conditions from Chapter 3 at the precise exit point of each encoded path. The remaining case is to ensure they also hold at exceptional exits.

Exceptions

So far we have provided a framework for checking safety conditions along complete execution sequences: given a sequence of instructions, we can assert that the safety condition holds when the sequence terminates as expected, typically at a `cret` or `mret`. This covers executions where the full code block runs to completion. What we have not yet considered are executions that end *prematurely*, namely when an instruction raises an exception. Since safety must hold at *any* exit point, we must extend the framework to cover these exceptional exits as well.

Encoding strategy. To cover exceptions, we extend the verification to every prefix of the code sequence. Whenever execution reaches instruction l_i in WB, we assert that either

- no exception is raised at this point (`~wbexc_err`), or
- the safety condition already holds.

This captures the intuition that an exception exit at instruction l_i must be just as safe as returning normally at the end of the sequence.

Formalization. In temporal logic, the property for prefix $l_0, \ldots, l_i$ can be expressed as:

$$wb_r(l_0) \wedge \mathbf{X} wb_r(l_1) \wedge \cdots \wedge \mathbf{X}^i wb(l_i) \rightarrow \mathbf{X}^i(\neg wbexc_err \vee condition)$$

Here, $wb_r(l) \equiv wb(l) \wedge ready$, and for simplicity we assume single-cycle steps (multi-cycle cases insert the appropriate number of $\mathbf{X}$ operators, as in Section 4.2). Note that for l_i, we use $wb(l_i)$ (not $wb_r(l_i)$), because we must also consider the case where an exception is raised at l_i, which would be excluded by *ready*. In SVA, this corresponds to reusing the existing straight-line encoding of the prefix, cut off at l_i, and adding the exception check on the right-hand side:

```
(property_prefix_up_to_li)
|->~wbexc_err |safety_condition;
```

Worked example. As a concrete illustration, consider the prefix of the previously discussed unsealer code fragment (see Section 4.2) that ends at instruction `l06_cgetbase`. The following property states: if execution reaches this point in WB, then either no exception occurs, or the two safety conjuncts `obj_ptr_safe` and `us_auth_safe` already hold:

```
property no_wbexc_err_3_l06_cgetbase_prop;
    (`INSTR_WB(l00_cgettag) &&instr_will_progress
    ##1 `INSTR_WB(l04_beqz) &&instr_will_progress &&~wbexc_has_branched
    ##1 `INSTR_WB(l06_cgetbase))
    |->~wbexc_err |(obj_ptr_safe &&us_auth_safe);
endproperty;
```

Such properties must be generated for every prefix of the code sequence; we automated this with a Python script to avoid boilerplate.

Together with the *sequence* $\rightarrow$ *condition* checks for normal returns, these prefix properties ensure that safety conditions hold at *all* exit points: both when execution returns to completion and when it diverts into an exception handler. This completes the framework for proving safety properties of binary execution on CHERIoT-Ibex.

Chapter 5

Verification Results on the Unsealer and Switcher

In the previous chapters we developed both the *safety conditions* to be proved (Chapter 3) and the *execution encoding framework* needed to reason about binaries on the CHERIoT-Ibex (Chapter 4). We now turn to the central question: do the unsealer and switcher modules of the CHERIoT RTOS satisfy their intended security guarantees when verified against the RTL model of the CHERIoT-Ibex?

Overview. This chapter presents the outcome of applying our methodology to these two case studies. For the unsealer, we report a complete verification: all normal and exceptional execution paths were checked against the safety conditions, revealing one previously unknown bug which we discuss in detail. For the switcher, we present partial results: a simplified code fragment was verified under additional assumptions, establishing safety for the capability stack pointer (`csp`). Alongside the verification outcomes, we also evaluate the practical feasibility of our approach, including timing results. In addition, Section 5.3 offers an exploratory discussion of how k-induction could be adapted to interval properties in this setting.

5.1 Complete verification of the unsealer

Concrete implementation of security properties

The unsealer's verification rests on two security conditions introduced in Section 2.2 and expanded in Section 3.2:

1. **Unsealing Authority Protection**: The global unsealing authority (`us_auth`) must not be present in the register file at any exit point from the unsealer. Neither `us_auth` nor any capability derived from it may be exposed.

2. **Controlled Sealed Object Exposure**: At any exit point, the sealed object pointer (`obj_ptr`) may only appear in:

   - register `ca1` in its original sealed form, or
   - register `ca0` in its correctly unsealed form, with the header removed. Any other appearance of `obj_ptr` or a derived capability in the register file violates safety.

Together, these claims characterize the **safety properties** of the unsealer. In practice, we work with the predicates `us_auth_safe` and `obj_ptr_safe` defined in Section 3.2, which capture these conditions precisely.

Paths through the unsealer. The unsealer binary (see Figure 5 in the appendix) consists of a single success path (see Figure 4 in the appendix) with a sequence of checks, each of which can branch to a short failure handler, encoded as:

```
sequence failure_sequence;
    (`INSTR_WB(l40_li_a2) &&instr_will_progress
    ##1 `INSTR_WB(l42_li_a0) &&instr_will_progress
    ##1 `INSTR_WB(l44_cret) );
endsequence
```

There are six such off-ramps. Together with the full success path, this yields seven execution paths to be verified. For each of these we constructed an SVA sequence following the methodology of Chapter 4. This ensured that every possible control-flow outcome of the binary was covered.

Branch properties. For each sequence, we constructed a property of the form:

```
property branch_2_safe_prop;
    (branch_2_l06_cgetbase ands assumption
    |->us_auth_safe &&obj_ptr_safe);
endproperty
```

This example corresponds to the branch at `l0a_bne`, directly following the check at `l06_cgetbase`. The sequence definition can be found in Section 4.2. The property states that if this path is taken under the pre-execution assumptions, then both safety conditions hold at exit. Analogous properties were generated for all six off-ramps and the full success path.

Exception properties. To cover exceptional exits, we generated one property for every prefix of the success path, as discussed in Section 4.3. The failure sequence was treated separately: here, we proved with a dedicated property that it cannot raise exceptions (all instructions only zero registers), so no additional exception properties were required for the branch paths. In total, 18 exception properties were generated for the 18 instructions of the success path.

Assumptions. The proofs relied on a single baseline assumption: at the entry point of the code sequence, no registers other than `ca0`, `ca1`, and `ca2` may hold any capability derivable from the two sensitive inputs `us_auth` and `obj_ptr`. This captures the intended RTOS usage, where these arguments are provided directly in their designated registers, and the rest of the register file is assumed clean: Stronger refined assumptions were introduced later, when analysing the discovered bug (see next subsection).

Completeness. With seven branch properties and 18 exception properties, the verification covered all 25 possible exit points of the unsealer binary. Together, they form a complete proof obligation: if all hold, then the unsealer satisfies its safety conditions at every normal and exceptional exit. All properties proved in this setup, except for one exception property, which produced a counterexample uncovering a previously unknown bug. We discuss this bug in the next subsection.

Raised issue and bug

One exception property failed during verification and revealed a subtle bug in the fast unsealer implementation. The property concerned the prefix ending at instruction `l28_clw`, where the unsealer attempts to load the software type tag from the just-unsealed pointer:

```
property no_wbexc_err_13_l28_clw_prop;
    (assumption and (`INSTR_WB(l00_cgettag) &&instr_will_progress
        ##1 ...
        ##1 `INSTR_WB(l28_clw) &&~instr_will_progress)
    |->(~wbexc_err |(obj_ptr_safe &&us_auth_safe)));
endproperty;
```

The counterexample showed that if the allocator provides a sealed capability whose length is < 16 bytes, the `clw` traps (bounds violation). At that point, register `ca2` holds the unsealed capability including its sensitive header, violating the safety condition `obj_ptr_safe`. This issue was raised with SCI and confirmed. As they summarized:

> If the allocator uses its loader-provided hardware sealing key to construct a sealed cap whose length is less than 16, then the fast unsealer's load of the virtual object type can trap. [...] as of that load instruction, ca2 is the unsealed pointer to the underlying allocation, which is not stellar. [13]

From a verification perspective, the property proves once we add the explicit assumption that the sealed object pointer `ca1` has load permission and appropriate

bounds (`ca1_assumption`), which reflects the intended preconditions for calling the unsealer.

Derived-capability assumptions. A separate question concerned possible relationships between the sensitive inputs (`obj_ptr` and `us_auth`) at entry. If one were derivable from the other, parts of the safety claims could become vacuous. SCI indicated that, in the intended RTOS usage, the arguments should be independent, though this may not be mechanically guaranteed in all contexts. For completeness, we therefore considered two regimes:

- **Baseline**: no registers other than `ca0` , `ca1` , and `ca2` holds any capability derived from `obj_ptr` or `us_auth` .
- **Independent-args**: in addition to the baseline, `obj_ptr` and `us_auth` are not derivable from each other; i.e. `ca1` must not contain a capability derived from `us_auth` and vice versa.

Both regimes were sufficient to prove the properties; the independent-args setting best reflects the intended system model. The implementation of the final assumption block, integrating the object-pointer well-formedness and argument independence, can be found in Figure 6 in the appendix.

Timing results

Beyond correctness, a central question is the *feasibility* of our verification workflow: can the safety properties be discharged within practical time bounds? To evaluate this, we measured solver runtimes for all unsealer properties under the assumptions described above. There results represent **unoptimized runs**: no advanced mechanisms such as SST tunnelling or cone-of-influence reduction were applied. In practice, such optimizations can often reduce proof times by a constant or even multiplicative factor.

The runtimes reported are wall-clock runtimes per property, but since all proofs were executed in parallel, the overall verification time is determined not by the sum but by the longest proof.

Exception properties. Each of the 18 exception properties (prefixes of the success path) discharged in under two minutes, with runtimes between 43 s and 57 s, except for one outlier at 163 s (see Table 5.1). This outlier corresponds exactly to the property at `l28_clw` , which had previously produced a counterexample. The

longer runtime is plausibly explained by the solver having to explore the possibility of an exception and check capability-derivability conditions before convergence.

Property	Time (s)
`no_wbexc_err_1_l00_cgettag`	43.7
`no_wbexc_err_2_l04_beqz`	57.6
...	...
`no_wbexc_err_13_l28_clw`	163.1
...	...
`no_wbexc_err_17_l36_sub`	57.6
`no_wbexc_err_18_l3a_csetbounds`	43.7

Table 5.1: Runtime of exception properties (excerpt).

Branch properties. The seven branch properties (success path plus six off-ramps) were all proven using k-induction (see Section 1.1). This is because properties of the form that we have been using ($(p_0 \wedge \mathbf{X}p_1 \wedge \cdots \wedge \mathbf{X}^n p_n) \rightarrow \mathbf{X}^n p_{safe}$) adapt easily to the k-induction formulas, as explored in Section 5.3. In every case, the solver chose a k exactly one larger than the number of cycles in the antecedent. This is natural: the antecedent spans n steps, and proving the implication requires induction at depth $n+1$ to capture the final exit state.

The runtimes show a clear near-linear scaling with the length of the antecedent sequence. The shortest off-ramp sequence (6 cycles) proved in ≈ 406 s, while a mid-length path of 13 cycles required ≈ 1800 s. Figure 5.1 plots runtime against sequence length for the first 6 branch properties, confirming this linear trend.

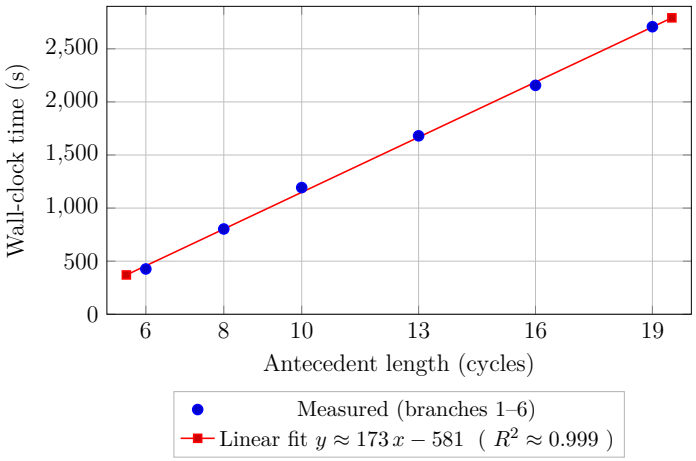


Figure 5.1: Runtime scaling of branch properties with antecedent length. Values can be found in Table 1 in the appendix. All proofs ran in parallel; overall wall-clock is the maximum single runtime.

Success path property. The only outlier is the full success path (20 cycles), which required ≈ 14480 s — substantially longer than the ≈ 2800 s suggested by the linear trend. The reason is likely structural: unlike the branch paths, which clear sensitive registers in the failure path, the success path retains and transforms them throughout execution. As a result, the solver must maintain capability-derivability constraints across a much larger portion of the trace, significantly inflating runtime. This suggests that the underlying linear scaling still holds, but with a larger constant factor in cases where sensitive values persist longer. In practice, established optimisation techniques (e.g. SST tunnelling) could reduce this constant substantially, bringing such proofs closer to the expected trend.

Takeaway. The timing results show that:

- Exception properties are cheap, proving in under 2 minutes.
- Branch properties scale linearly with path length.
- All properties are provable overnight (max. 4h) even without optimization.
- This linearity is a strong indicator that the WB-stage encoding is tractable. With solver tuning and engineering effort, we expect runtimes could be improved by a constant factor.

5.2 Partial results on the switcher

Setup and simplifications

Scope. The complete verification of the switcher is beyond the scope of this thesis. The switcher is significantly more complex than the unsealer, and a full treatment would require reasoning about the provenance of values, constructing a faithful memory model for values read from the trusted stack, and handling the switcher's intricate control flow. However, having established our methodology on the unsealer, we can already explore a simplified fragment of the switcher to demonstrate how the approach generalizes. Importantly, this limitation is not methodological, but practical: with appropriate support for memory modelling and provenance reasoning, the same WB-stage encoding could in principle be scaled to the full switcher.

Simplifications. The exploration concerns a fragment of the context restore functionality of the switcher, which is responsible for restoring the state at the end of a context switch (see Section 2.2). The only sensitive capability here is `csp`. For this first exploration, we apply several simplifications to make the verification task

tractable: the capability stack pointer `csp` is assumed valid with sufficient bounds (so loads will not trap); the guard branches in the code are assumed not taken. These simplifications eliminate code paths and reduce the number of restore instructions that must be checked. In practice, this boils the context-restore code down to a limited straight-line fragment:

```
cspecialw mtdc, csp
clc ct2, TrustedStack_offset_mepcc(csp)
clw ra, TrustedStack_offset_mstatus(csp)
csrw mstatus, ra
cspecialw mepcc, ct2
clc cra, TrustedStack_offset_cra(csp)
clc ct2, TrustedStack_offset_ct2(csp)
clc csp, TrustedStack_offset_csp(csp)
mret
```

The verification challenge posed by SCI was:

> Prove that this block of instructions results in the at-entry value of `csp` persisting in `mtdc`, assuming that non-`csp` registers are zero at entry, the general-purpose register file does not hold a capability that could be derived from `csp`'s at-entry value.

Assumptions and safety conditions

At entry, the only sensitive value is the capability stack pointer `csp`. let `csp_at_entry` denote its at-entry value. The following assumptions capture the intended execution environment on this code fragment:

- **Validity of `csp`**: The capability `csp` is valid, has sufficient bounds and includes load permission, so all memory accesses through it succeed. This is summarized by the predicate `csp_assumptions`.
- **Independence**: Initially, we assumed that all non-`csp` registers are zero, but this can be relaxed to: at entry, no capability register apart from `cra`, `ct2` and `csp` holds any capability derivable from `csp`.
- **Memory independence**: All capabilities loaded from memory during the block are assumed not to be derivable from `csp_at_entry`.
- **System register access**: We assume the switcher executes with sufficient privilege to access system registers, so writes to `mstatus`, `mepcc`, and `mtdc` do not trap.

These entry-state assumptions are implemented directly in SVA and asserted once at the start of the sequence (see Figure 7 in the appendix).

The memory-independence assumption, by contrast, must hold not just at the start but throughout execution: every capability fetched from memory must remain independent of `csp_at_entry`. To capture this, we implement a derivability check on the incoming LSU capability, asserted at each step of the code sequence. The following SVA snippet shows how this invariant is encoded.

```
logic csp_at_entry_no_memory_overlap =~derived_from(lsu_cap_i, lsu_addr_i, csp_at_entry,
    ↪ csp_addr_at_entry);

// usage (where code_sequence is the previously defined code sequence):
sequence memory_independence;
    csp_at_entry_no_memory_overlap throughout code_sequence;
endsequence
```

As defined in Section 3.2, the verification target is:

1. **Persistence of** `csp_at_entry`: The at-entry value of `csp` is successfully installed in the system register `mtdc`.
2. **Non-leakage of** `csp_at_entry`: No register in the capability register file may contain a capability derivable from `csp_at_entry`.

Verification and results

Property encoding. The straight-line fragment of line 1-8 was encoded as w WB-stage sequence with explicit stall delays. The property asserts that if the sequence executes under the assumptions, then `mtdc` equals the at-entry `csp` and no other register contains a derivable capability. As with the unsealer, the engine used was k-induction, with k one greater than the number of cycles in the sequence. The property discharged successfully and thus establishes the quarantine of `csp` in `mtdc`.

Timing. The full 8 line sequence (with antecedent length of 15 cycles due to a number of stalls) proved in approximately 180 s — significantly faster than the unsealer due to lesser complexity. We also checked all prefixes of the sequence with correspondingly adapted safety conditions and observed that runtimes followed an approximately linear trend with sequence length.

Takeaway. This case study confirms that the WB-stage encoding methodology can be applied to the switcher as well. The absence of branching and exceptions made proofs straightforward, but scaling to the full switcher will require richer infrastructure for memory modelling, provenance reasoning and control-flow coverage.

5.3 On k-induction for interval properties

Note: The material in this section is exploratory. The verification tool used in this project (Jasper) applies proprietary proof engines internally. While the specific proofs in this work were discharged by a k-induction engine, its exact implementation is not accessible. The following material therefore does not describe the tools internal workings, but presents an adaptation of the k-induction principle to interval properties, as a mathematical illustration.

In this thesis, we have mostly considered properties of the form

$$\varphi_n \equiv \left(p_0 \wedge \mathbf{X}p_1 \wedge \cdots \wedge \mathbf{X}^{n-1}p_{n-1}\right) \rightarrow \mathbf{X}^n p_s$$

where p_i are state predicates and p_s is the safety predicate. We call this an *interval property of length n*. Our aim is to prove global properties of the form $\mathbf{G}\varphi_n$: on every execution path, whenever an interval meets the antecedent, the safety condition p_s must hold at its end.

Interval properties are not only useful in software verification, but also especially well-suited to k-induction, since their truth depends on fixed-length intervals rather than isolated states. However, most standard definitions of k-induction apply only to state invariants, so we develop the necessary adaptation here.

Intervals and local semantics

Throughout this section we consider a Kripke structure $M = (S, S_0, R, AP, L)$ as described in Section 1.1 with the representation of S_0, R and $p \in AP$ as first-order predicates.

We define an *interval of length m* to be a sequence of $m+1$ states that is connected by the transition relation (see Definition 5).

Definition 5 (Interval) *An* interval I of length m *(or* m-interval*) is an* $(m+1)$*-tuple of states* $(s_0, \ldots, s_m)$ *such that* $R(s_i, s_{i+1})$ *holds for all* $0 \leq i < m$*.*

- *The interval is* initial*, if* s_0 *is an initial state.*
- *It is* reachable *if it appears as a consecutive subsequence of some path* $\pi = (s_j)_{j\in\mathbb{N}}$ *(see Equation* (1.1)*)*
- *A* subinterval *of* I *is any subsequence* $(s_a, \ldots, s_b)$ *with* $0 \leq a \leq b \leq m$*.*

Localization. By a mild abuse of notation, we also write φ_n for the local satisfaction relation that we define on n-intervals $(s_i, \ldots, s_{i+n})$:

$$\varphi_n(s_i, \ldots, s_{i+n}) \equiv \left(\bigwedge_{j=0}^{n-1} p_j(s_{i+j}) \right) \rightarrow p_s(s_{i+n}) \tag{5.1}$$

Its negation expands to

$$\neg\varphi_n(s_i, \ldots, s_{i+n}) \equiv \bigwedge_{j=0}^{n-1} p_j(s_{i+j}) \wedge \neg p_s(s_{i+n}) \tag{5.2}$$

Here, $\varphi_n(s_i, \ldots, s_{i+n})$ captures the condition on a single interval, while the LTL property $\mathbf{G}\varphi_n$ requires that this condition holds on *every reachable n-interval* in the model.

Applying k-induction

To apply k-induction, note that an interval property of length n requires $n+1$ states, so we need $k \geq n+1$. The proof follows the same structure as classical k-induction (see Section 1.1), but with intervals replacing single states.

- **Base**: Any initial interval of length n satisfies φ_n:

$$S_0(s_0) \wedge \bigwedge_{i=0}^{n-1} R(s_i, s_{i+1}) \wedge \neg\varphi(s_0, \ldots, s_n) \quad \text{is unsatisfiable}$$

- **Step**: If in an interval of length $n+1$, the first subinterval of length n satisfies φ_n, then the shifted subinterval also satisfies φ_n:

$$\bigwedge_{i=0}^{n} R(s_i, s_{i+1}) \wedge \varphi(s_0, \ldots, s_n) \wedge \neg\varphi(s_1, \ldots, s_{n+1}) \quad \text{is unsatisfiable}$$

Expanded into state predicates (applying equations (5.1) and (5.2)), we obtain:

- **Base**:

$$S_0(s_0) \wedge \bigwedge_{i=0}^{n-1} R(s_i, s_{i+1}) \wedge \bigwedge_{i=0}^{n-1} p_i(s_i) \wedge \neg p_s(s_n) \quad \text{is unsatisfiable}$$

- **Step**:

$$\bigwedge_{i=0}^{n} R(s_i, s_{i+1}) \wedge \left(\bigwedge_{i=0}^{n-1} p_i(s_i) \rightarrow p_s(s_n) \right) \wedge \left(\bigwedge_{i=1}^{n} p_i(s_i) \wedge \neg p_s(s_{n+1}) \right) \quad \text{is unsatisfiable}$$

Why this proves $\mathbf{G}\varphi_n$. If both the base and step formulas are unsatisfiable, then φ_n holds on every reachable interval of length n.

- The **base case** guarantees that the initial interval $(s_0, \ldots, s_n)$ of any path satisfies φ_n.
- The **step case** guarantees that if an interval $(s_i, \ldots, s_{i+n})$ of a path satisfies φ_n, then the shifted interval $(s_{i+1}, \ldots, s_{i+n+1})$ does as well, so satisfaction moves forward along the path.

Therefore, starting from the base interval of a path, satisfaction propagates inductively to all subsequent intervals. Since this holds for every path, all reachable intervals satisfy φ_n, i.e. the global truth condition $\mathbf{G}\varphi_n$.

Generalization to $k \geq n+1$. While k-induction requires $k \geq n+1$ to cover intervals of length n, larger k can also be used. The scheme is:

- **Base**: No initial interval of length $k-1$ contains a n-subinterval violating φ_n:

$$S_0(s_0) \wedge \bigwedge_{i=0}^{k-2} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k-n-1} \neg\varphi(s_i, \ldots, s_{i+n}) \quad \text{is unsatisfiable}$$

- **Step**: In any interval of length k, if all but the last n-subinterval satisfy φ_n, then so must the last:

$$\bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigwedge_{i=0}^{k-n-1} \varphi(s_i, \ldots, s_{i+n}) \wedge \neg\varphi(s_{k-n}, \ldots, s_k) \quad \text{is unsatisfiable}$$

Conclusion. This construction adapts k-induction from state invariants to temporal interval properties. The essential change is that the inductive invariant is not a predicate on single states but on *intervals of states*. While exploratory, this illustrates how temporal reasoning about bounded execution fragments can be expressed in an inductive verification framework.

Chapter 6

Conclusion and prospects

This thesis addressed how to formally verify that sensitive capabilities in CHERIoT-Ibex binaries are not exposed or misused during execution. It developed and evaluated a methodology that expresses safety properties directly over binary execution paths and proves them against the RTL implementation of the processor.

The central contribution is a *systematic WB-stage encoding framework*. By observing effects at the writeback stage, the framework supports temporal properties across straight-line, branching, and exceptional execution, while also tracking at-entry values of sensitive capabilities. This yields a tractable yet expressive verification methodology.

Two case studies demonstrated the framework's usefulness:

- **Unsealer**: all normal and exceptional paths were verified, uncovering a subtle bug in the fast unsealer implementation, later confirmed by SCI Semiconductor.
- **Switcher**: a simplified fragment of the context-restore procedure was verified under assumptions about memory and control flow, suggesting that the approach generalises to more complex components.

Finally, the work established *feasibility*: all properties discharged with k-induction, proof times scaled linearly with path length and even the longest proofs completed within hours. To our knowledge, this is the first demonstration of verifying binaries directly against a cycle-accurate RTL model, closing the compilation gap and providing unusually strong end-to-end guarantees for capability safety.

The results point to several natural extensions:

- **Automation**: properties were constructed by hand; automating their generation from binaries would eliminate the main bottleneck and enable larger-scale verification.

- **Proof optimisation**: while runtimes were tractable, established techniques such as SST tunnelling could further reduce solver cost and enable subsystem-scale verification.
- **Generality**: the methodology is not limited to CHERIoT-Ibex: it could be applied to other CHERI processors or more broadly wherever binary-level reasoning is required.

Beyond the specific results on the unsealer and switcher, this work points to a broader possibility: that security mechanisms in compartmentalised systems can be not only designed but *formally proven* end-to-end, from cycle-accurate hardware through to system binaries. With further progress in automation and optimisation, such methods could raise the assurance bar for real-world systems well beyond current practice.

References

[1] A. Robison, "Anatomy of a Firmware Attack - Eclypsium | Supply Chain Security for the Modern Enterprise," *Eclypsium | Supply Chain Security for the Modern Enterprise*, Dec. 2019. [Online]. Available: https://eclypsium.com/threat-reports/anatomy-of-a-firmware-attack-2

[2] "CWE - 2024 CWE Top 25 Most Dangerous Software Weaknesses," Nov. 2024, [Online; accessed 4. Aug. 2025]. [Online]. Available: https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html

[3] A. Cudmore, "Current and Future Flight Operating Systems," May 2007, [Online; accessed 5. Aug. 2025]. [Online]. Available: https://ntrs.nasa.gov/citations/20080040872

[4] V. D'Silva, D. Kroening, and G. Weissenbacher, "A Survey of Automated Techniques for Formal Software Verification," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 27, no. 7, pp. 1165–1178, Jul. 2008.

[5] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an OS kernel," in *ACM Conferences*. New York, NY, USA: Association for Computing Machinery, Oct. 2009, pp. 207–220.

[6] S. Amar, D. Chisnall, T. Chen, N. W. Filardo, B. Laurie, K. Liu, R. Norton, S. W. Moore, Y. Tao, R. N. M. Watson, and H. Xia, "CHERIoT: Complete Memory Safety for Embedded Devices," in *ACM Conferences*. New York, NY, USA: Association for Computing Machinery, Oct. 2023, pp. 641–653.

[7] microsoft, "cheriot-ibex," Aug. 2025, [Online; accessed 6. Aug. 2025]. [Online]. Available: https://github.com/microsoft/cheriot-ibex/blob/main/README.md

[8] D. Chisnall, "Why did you write a new RTOS for CHERIoT?" *CHERIoT Platform*, Oct. 2024. [Online]. Available: https://cheriot.org/rtos/philosophy/history/2024/10/24/why-new-rtos.html

[9] Cheriot-Platform, "cheriot-rtos," Aug. 2025, [Online; accessed 6. Aug. 2025]. [Online]. Available: https://github.com/CHERIoT-Platform/cheriot-rtos/blob/main/README.md

[10] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: revisiting RISC in an age of risk," in *ACM SIGARCH Computer Architecture News*. New York, NY, USA: Association for Computing Machinery, Jun. 2014, vol. 42, no. 3, pp. 457–468.

[11] D. Chisnall, "Introducing sealed types," *CHERIoT Platform*, Jan. 2025. [Online]. Available: https://cheriot.org/sealing/compiler/2025/01/30/introducing-sealed-types.html

[12] "SCI Semiconductor - Home," Aug. 2025, [Online; accessed 26. Aug. 2025]. [Online]. Available: https://www.scisemi.com

[13] Cheriot-Platform, "cheriot-rtos," Jul. 2025, [Online; accessed 6. Aug. 2025]. [Online]. Available: https://github.com/CHERIoT-Platform/cheriot-rtos/issues/550

[14] "Cadence | Computational Software for Intelligent System Design," Aug. 2025, [Online; accessed 26. Aug. 2025]. [Online]. Available: https://www.cadence.com/en_US/home.html

[15] Tita Rosemeyer, "cheriot-ibex," Sep. 2025, [Online; accessed 4. Sep. 2025]. [Online]. Available: https://github.com/TitaRosemeyer/cheriot-ibex

[16] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, Apr. 2008, https://web.eecs.umich.edu/~movaghar/Principles%20of%20Model%20Checking-Book-2008.pdf.

[17] D. Kroening, D. Peled, E. M. Clark Jr, H. Veith, and O. Grumberg, *Model Checking, second edition (Cyber Physical Systems Series)*. The MIT Press, Dec. 2018.

[18] "Department of Computer Science and Technology: Capability Hardware Enhanced RISC Instructions (CHERI)," Oct. 2024, [Online; accessed 17. Aug. 2025]. [Online]. Available: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/

[19] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Commun. ACM*, vol. 9, no. 3, p. 143–155, Mar. 1966. [Online]. Available: https://doi.org/10.1145/365230.365252

[20] H. Levy, *Capability-based Computer Systems*. Digital Press, 1984. [Online]. Available: https://books.google.de/books?id=dQMnAAAAMAAJ

[21] "Department of Computer Science and Technology: CHERI Frequently Asked Questions (FAQ)," Oct. 2020, [Online; accessed 16. Aug. 2025]. [Online]. Available: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-faq.html

[22] "Redesigning Hardware to Support Security: CHERI," Feb. 2023, [Online; accessed 17. Aug. 2025]. [Online]. Available: https://www.usenix.org/publications/loginonline/redesigning-hardware-support-security-cheri

[23] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, H. Almatary, J. Anderson, J. Baldwin, G. Barnes, D. Chisnall, J. Clarke, B. Davis, L. Eisen, N. W. Filardo, R. Grisenthwaite, A. Joannou, B. Laurie, A. T. Markettos, S. W. Moore, S. J. Murdoch, K. Nienhuis, R. Norton, A. Richardson, P. Rugg, P. Sewell, S. Son, and H. Xia, "Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-951, oct 2020. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-951.pdf

[24] "CHERIoT Platform," Apr. 2025, [Online; accessed 19. Aug. 2025]. [Online]. Available: https://cheriot.org/

[25] S. Amar, T. Chen, D. Chisnall, F. Domke, N. Filardo, K. Liu, R. Norton-Wright, Y. Tao, R. N. M. Watson, and H. Xia, "CHERIoT: Rethinking security for low-cost embedded systems," *Microsoft Research*, Feb. 2023. [Online]. Available: https://www.microsoft.com/en-us/research/publication/cheriot-rethinking-security-for-low-cost-embedded-systems

[26] Cheriot-Platform, "cheriot-rtos," Mar. 2023, [Online; accessed 19. Aug. 2025]. [Online]. Available: https://github.com/CHERIoT-Platform/cheriot-rtos/blob/main/docs/architecture.md

[27] L.-E. Ploix, A. Armstrong, T. Melham, R. Lin, H. Wang, and A. Courtney, "Comprehensive Formal Verification of Observational Correctness for the CHERIoT-Ibex Processor," *arXiv*, Feb. 2025.

[28] Cheriot-Platform, "cheriot-rtos/sdk/core/token_library/token_unseal.S," Sep. 2025, [Online; accessed 1. Sep. 2025]. [Online]. Available: https://github.com/CHERIoT-Platform/cheriot-rtos/blob/main/sdk/core/token_library/token_unseal.S

Appendix

Supplementary Material

All SystemVerilogAssertions code, properties, and scripts developed for this thesis are available in the accompanying repository at `https://github.com/TitaRosemeyer/cheriot-ibex` [15].

The repository contains the complete verification environment, including the CHERIoT-Ibex RTL code base, the follower module, property encodings, and the case study for the unsealer and switcher.

Code Fragments

Figures 1-7 collect the main code listings and encodings referenced in Chapters 3-5. Table 1 holds the reference values for the proof times referenced in Chapter 5.

Branch property	Antecedent length (cycles)	Time (s)
`branch_1_safe`	6	426.5
`branch_2_safe`	8	803.1
`branch_3_safe`	10	1193.1
`branch_4_safe`	13	1680.5
`branch_5_safe`	16	2155.7
`branch_6_safe`	19	2708.1

Table 1: Reference values for Figure 5.1.

```
function automatic bit has_fewer_perms(reg_cap_t cap1, reg_cap_t cap2);
    // Check if cap1 has at least as strict permissions as cap2
    logic [11:0] perms1 =expand_perms(cap1.cperms);
    logic [11:0] perms2 =expand_perms(cap2.cperms);

    return (perms1 &perms2) ==perms1;
endfunction
```

```
function automatic bit has_stricter_bounds(reg_cap_t cap1, logic [31:0] addr1, reg_cap_t cap2, logic [
    ↪ 31:0] addr2);
    // Check if cap1 has at least as strict bounds as cap2
    logic [32:0] top1 =get_bound33(cap1.top, cap1.top_cor, cap1.exp, addr1);
    logic [32:0] top2 =get_bound33(cap2.top, cap2.top_cor, cap2.exp, addr2);
    logic [32:0] base1 =get_bound33(cap1.base, {2{cap1.base_cor}}, cap1.exp, addr1);
    logic [32:0] base2 =get_bound33(cap2.base, {2{cap2.base_cor}}, cap2.exp, addr2);

    return (top1 <=top2) &&(base1 >=base2);
endfunction
```

```
function automatic bit derived_from(reg_cap_t cap1, logic [31:0] addr1, reg_cap_t cap2, logic [31:0]
    ↪ addr2);
    // Check if cap1 is derivable from cap2
    logic both_valid =(cap1.valid &&cap2.valid);
    logic fewer_perms =has_fewer_perms(cap1, cap2);
    logic stricter_bounds =has_stricter_bounds(cap1, addr1, cap2, addr2);

    return both_valid &&stricter_bounds &&fewer_perms;
endfunction
```

Figure 1: Implementation of derivability as described in defintion 4 in SystemVerilogAssertions (SVA)

```
`define MAKE_VAR_AT_ENTRY(VARNAME, VARTYPE, TRIGGERINSTR) \
    VARTYPE VARNAME``_at_entry; \
    always_latch begin \
        if (($past(instr_will_progress) &&`INSTR ==TRIGGERINSTR &&wbexc_exists) ||~rst_ni) begin \
            VARNAME``_at_entry =VARNAME; \
        end \
    end
```

Figure 2: Macro for creation of at-entry snapshots of variables

```
`define INSTR `CR.instr_rdata_id

assign ex_success =`ID.instr_done;
assign ex_err =`IDC.exc_req_d;
assign ex_kill =`ID.wb_exception |~`ID.controller_run;
// Note that this only kills instructions because e.g. of a jump ahead of it or an exception

assign exc_finishing =`IDC.ctrl_fsm_cs ==`IDC.FLUSH;
assign wbexc_handling_irq =`IDC.ctrl_fsm_cs ==`IDC.IRQ_TAKEN;
assign wb_finishing =wbexc_is_wfi? wfi_will_finish:`CR.instr_done_wb;
assign wfi_will_finish =`IDC.ctrl_fsm_cs ==`IDC.FLUSH;
assign wbexc_err =wbexc_ex_err |
                  `IDC.wb_exception_o |
                  ((`IDC.ctrl_fsm_cs ==`IDC.FLUSH) &~wbexc_csr_pipe_flush);

assign wbexc_finishing =wbexc_exists &(wbexc_err ?exc_finishing :wb_finishing);

assign instr_will_progress =(~wbexc_exists |wbexc_finishing) &~ex_kill &(ex_success |ex_err);

always_comb begin
    if (`CR.instr_new_id) begin
        ex_has_branched_d =1'b0;
    end else begin
        ex_has_branched_d =ex_has_branched_q;
    end
    ex_has_branched_d =ex_has_branched_d |(`IF.branch_req &&~ex_kill &&`IDC.ctrl_fsm_cs ==`IDC.DECODE);
end

always @(posedge clk_i or negedge rst_ni) begin
    if (~rst_ni) begin
        wbexc_exists <=1'b0;
        idex_has_compressed_instr <=1'b0;
        ex_has_branched_q <=1'b0;
        wbexc_csr_pipe_flush <=1'b0;
    end else begin
        if (wbexc_finishing) begin
            wbexc_exists <=1'b0;
        end

        ex_has_branched_q <=ex_has_branched_d;
        if (instr_will_progress) begin
            ex_has_branched_q <=1'b0;
            wbexc_has_branched <=ex_has_branched_d;
            wbexc_post_wX <=spec_post_wX;
            wbexc_post_wX_addr <=spec_post_wX_addr;
            wbexc_post_wX_en <=spec_post_wX_en;

            ...

            wbexc_instr <=idex_compressed_instr;
            wbexc_decompressed_instr <=`CR.instr_rdata_id;
            wbexc_compressed_illegal <=`CR.illegal_c_insn_id;
            wbexc_exists <=1'b1;
            wbexc_ex_err <=ex_err;
            wbexc_fetch_err <=`ID.instr_fetch_err_i;
            ...
        end

        if (`IF.if_id_pipe_reg_we) begin
            idex_compressed_instr <=`IF.if_instr_rdata;
            idex_has_compressed_instr <=1'b1;
        end
    end
end
```

Figure 3: The follower module implemented by Louis-Emile Ploix [27] (abridged)

```
sequence branch_7_success;
    (assumption and (`INSTR_WB(l00_cgettag) &&instr_will_progress
        ##1 `INSTR_WB(l04_beqz) &&instr_will_progress &&~wbexc_has_branched
        ##1 `INSTR_WB(l06_cgetbase) &&instr_will_progress
        ##1 `INSTR_WB(l0a_bne) &&instr_will_progress &&~wbexc_has_branched
        ##1 `INSTR_WB(l0e_cgetlen) &&instr_will_progress
        ##1 `INSTR_WB(l12_beqz) &&instr_will_progress &&~wbexc_has_branched
        ##1 `INSTR_WB(l14_cgetperm) &&instr_will_progress
        ##1 `INSTR_WB(l18_andi) &&instr_will_progress
        ##1 `INSTR_WB(l1c_bne) &&instr_will_progress &&~wbexc_has_branched
        ##1 `INSTR_WB(l1e_cunseal) &&instr_will_progress
        ##1 `INSTR_WB(l22_cgettag) &&instr_will_progress
        ##1 `INSTR_WB(l26_beqz) &&instr_will_progress &&~wbexc_has_branched
        ##1 `INSTR_WB(l28_clw) &&~instr_will_progress
        ##1 instr_will_progress
        ##1 `INSTR_WB(l2a_bne) &&instr_will_progress &&~wbexc_has_branched
        ##1 `INSTR_WB(l2e_cgettop) &&instr_will_progress
        ##1 `INSTR_WB(l32_cinoffset) &&instr_will_progress
        ##1 `INSTR_WB(l36_sub) &&instr_will_progress
        ##1 `INSTR_WB(l3a_csetboundsexact) &&instr_will_progress
        ##1 `INSTR_WB(l3e_cret))
        );
endsequence;
```

Figure 4: The encoding of the success path of the unsealer in SVA

```
...
.Ltoken_unseal_internal:
  /*
   * Register allocation:
   * - ca0 holds the user's sealing key, and is replaced with the unsealed
   * value or NULL
   * - ca1 holds the user's sealed object pointer
   * - ca2 holds the unsealing authority and is clobbered on failure
   * explicitly and on success with a scalar (the sealed payload's length)
   * - a3 is used within each local computation and never holds secrets
   */

  /* Verify key tag */
  cgettag a3, ca0
  beqz a3, .Lexit_failure

  /* Verify key address == base and len > 0 */
  cgetbase a3, ca0
  bne a0, a3, .Lexit_failure // as-integer access to ca0 gives address
  cgetlen a3, ca0
  beqz a3, .Lexit_failure

  /* Verify key has unseal permission */
  cgetperm a3, ca0
  andi a3, a3, CHERI_PERM_UNSEAL
  beqz a3, .Lexit_failure

  /* Unseal, clobbering authority */
  cunseal ca2, ca1, ca2

  /* Verify tag of unsealed form */
  cgettag a3, ca2
  beqz a3, .Lexit_failure

  /*
   * Load software type tag. This will not trap, thanks to above tag check and
   * because IRQs are deferred (see our export entry below)
   */
  clw a3, TokenSObj_offset_type(ca2)

  /* Verify that the loaded value matches the address of the key. */
  bne a0, a3, .Lexit_failure

  /* Subset bounds to ->data */
  // Get the top into a3
  cgettop a3, ca2
  // Move the address to the start of the data, clobber the user's sealing key
  cincoffset ca0, ca2, TokenSObj_offset_data
  // Subtract the address of the (to-be-returned-unsealed) data from the top to
  // give the length, clobbering our unsealing key.
  sub a2, a3, a0
  // Set the new bounds, using an exact setting so that any errors in the
  // allocator's alignment turn into an untagged capability here.
  csetboundsexact ca0, ca0, a2

  /* And that's an unwrap. */
  cret

.Lexit_failure:
  /* Failure; clobber potential sensitive state in ca2 and return null */
  zeroOne a2
  zeroOne a0
  cret

...
```

Figure 5: Implementation of the unsealer in token_unseal.S (abridged) [28]

```
// check object pointer assumptions
logic [11:0] ca1_has_load =expand_perms(ca1.cperms)[5]; // LD permission
logic [32:0] ca1_top =get_top_bound33(ca1, a1);
logic [31:0] ca1_base =get_base_bound32(ca1, a1);
logic ca1_clw_in_bounds =((ca1_base <=a1) &&(a1 <=(ca1_top -32'h4)) &&(ca1_top >=32'h4));
logic ca1_assumption =ca1_has_load &&ca1_clw_in_bounds;

// assume that the unsealing authority and object pointer are not leaked at the beginning of the
    ↪ execution
logic assumption;
always_comb begin
    if (`ASSUME_INDEPENDENT_ARGS) begin
        assumption =ca1_assumption &&no_derivatives_except_ca2_fn(ca2, a2) &&
            ↪ no_derivatives_except_ca1_fn(ca1, a1);
    end else begin
        // assume that the two sensitive input arguments are not derived from each other
        assumption =ca1_assumption &&no_derivatives_except_ca0_ca1_ca2_fn(ca2, a2) &&
            ↪ no_derivatives_except_ca0_ca1_ca2_fn(ca1, a1);
    end
end
```

Figure 6: Final assumption block for the unsealer verification

```
// ensure well-formedness of csp
logic csp_unsealed =csp.otype ==3'b000;
logic csp_has_permit_load =csp.cperms[4:3] ==2'b11 |csp.cperms[4:2] ==3'b101 |csp.cperms[4:1] ==4'b1001
    ↪  |csp.cperms[4:3] ==2'b01;
logic csp_addr_in_bounds =(csp_addr >=csp_base) &&(csp_addr +32'h88 +32'h8 <=csp_top);
logic csp_addr_aligned =(csp_addr %8 ==0);
logic csp_assumptions =csp.valid &&csp_unsealed &&csp_has_permit_load &&csp_addr_in_bounds &&
    ↪ csp_addr_aligned;

// Ensure system register access
logic PCCHasASR =(pcc.cperms[4:2] ==3'b011);

//combine all assumptions
logic assumption =PCCHasASR &&csp_assumptions &&no_derivatives_except_cra_ct2_csp(csp, csp_addr)
```

Figure 7: Implementation of the assumption for the switcher