Verifying the seL4 Core Platform

Verified mapping of seL4CP spec to CapDL Step 1: SDF Specification and Verification Approach

Zoltan A. Kocsis

December 2022

1 Introduction

The **seL4 Core Platform** (seL4CP) is a new operating system for embedded systems, based on the high-assurance seL4 operating system kernel. Unlike previous frameworks, such as CAmkES, which require a deep understanding of seL4 systems development, seL4CP has ease-of-deployment as its core aim, and is intended to be accessible to all embedded-systems developers.

The seL4 Core Platform was co-developed with a device called Laot that protects critical infrastructure (such as remotely-managed power stations) from cyber attacks. The Laot device was build using seL4CP by an engineer without prior seL4 systems development experience, which indicates that the usability aim had been successfully achieved.

CapDL is a language for describing access rights in seL4-based systems. CapDL specifications can be used to track which objects and entities have access to which seL4 capabilities, and provide complete descriptions of the capability distribution in a system running on the seL4 kernel. Moreover, CapDL specifications can be used as input to security analysis tools, including tools that help automate system bootstrapping processes. This makes CapDL an extremely powerful and versatile tool for managing seL4-based systems.

The Trustworthy Systems group, during the course of the DARPA Cyber Assured Systems Engineering (CASE) research project, has developed a *formally verified system initialiser* (`case-init`) that can place an seL4 system into any CapDL-specified state.

Since the CAmkES framework has a verified mapping from its architecture description language to CapDL, CAmkES projects can take advantage of the assured system initialisation provided by `case-init`. However, the System Description Format used by the Core Platform currently lacks an analogous CapDL mapping, so these benefits are not yet available to projects targeting seL4CP.

The goal of this project is to develop a verified mapping from the **SDF** format used by seL4CP to CapDL, so that seL4CP-based systems can take advantage of the assured system initialisation provided by system initialiser developed as part of the CASE project. This will requires developing new tooling to generate the necessary CapDL specifications from Core Platform system descriptions.

2 Rigorous Specification

2.1 Preamble

The seL4 Core Platform introduces a limited set of straightforward abstractions that closely mirror the objects of the seL4 microkernel, resulting in a streamlined implementation. A system implemented on seL4CP is composed of only three types of first-class platform objects, and the explicitly enumerated relationships between them. These first-class objects are protection domains (PDs), IRQs and memory regions (MRs). PDs can interact with each other through various properties: one such property is the presence of a channel, which serve as a means of authorizing communications in the system specification. Interactions through channels can occur in two forms: sending notifications, which provide asynchronous signaling, and making protected procedure calls (PPCs), which allow for synchronous function calls between different PDs.

2.2 Abstract Systems

In what follows, we fix a positive integer $k > 0$ called the *page size*, finite set $I \subseteq \mathbb{N}$ called the *set of IRQs*, a set $A \subseteq k\mathbb{N}$ called the *memory addresses*, and a finite set $\mathcal{V} \subseteq \{0, \ldots, 62\}$ of *valid protection domain identifiers*.

Definition: A *permission* is a subset of $\{\mathtt{W}, \mathtt{X}, \mathtt{C}\}$ where `W`, `C` and `X` are all distinct symbols. The set of permissions is denoted W.

Definition: A *priority* is an integer between 0 and 254. The set of priorities is denoted P. A budget-period is a pair of integers (x, y) such that $0 < x \leq y < 2^{64}$. The set of budget-periods is denoted B.

Definition: A *channel identifier* is an integer between 0 and 62. Let C denote the set of channel identifiers. An *end* (or *pdch*) is an element of the set $\mathcal{V} \times C$. A *comm* is an unordered pair $\{(v_1, c_1), (v_2, c_2)\}$ consisting of two ends such that $v_1 \neq v_2$.[1]

Definition: A *mapped memory region* consists of the following data:

- a *base PD* $b \in V$,
- a *physical address* $a_p \in A$,
- a *virtual address* $a_v \in A$,
- a *permission* $w \in W$,
- a size $n \in k\mathcal{N}$

so that for all $m < n$, $a_p + m \in A$ and $a_v + m \in A$ both hold.

[1] Forbidding self-channels here greatly simplifies concurrency handling in Activity 3.1.2.

Definition An *abstract system* consists of the following data:

- a set $\mathcal{V}_p \subseteq \mathcal{V}$ of *protected procedure call domains*,
- a set $\mathcal{C}$ of comms, called the *set of channels*,
- a set $\mathcal{I} \subseteq I \times \mathcal{V} \times C$ called the *set of IRQ mappings*,
- a finite set $\mathcal{M}$ of mapped memory ranges,
- a *priority map* $v_p : \mathcal{V} \to P$, and
- a *budget-period map* $v_b : \mathcal{V} \to B$.

subject to the conditions below:

1. All channels are disjoint, i.e. if $x, y \in \mathcal{C}$, then either $x = y$ or $x \cap y = \emptyset$.
2. IRQ ends and channel ends are disjoint, i.e. if $(i, e) \in \mathcal{I}$ and $x \in \mathcal{C}$, then $e \notin x$.
3. Virtual addresses are disjoint, i.e. if two mapped memory regions $x, y \in \mathcal{M}$ have the same base PD $b(x) = b(y)$, then either $a_v(x) < a_v(x) + n(x) \leq a_v(y) < a_v(y) + n(y)$ or else $a_v(y) < a_v(y) + n(y) \leq a_v(x) < a_v(x) + n(x)$.

2.3 Implementation Relation

Take an abstract system as defined above. Consider a finite set $\mathbb{V}$ of ELF binaries, and an assignment $i : V \to \mathbb{V}$. We say that (the capability distribution of) an seL4 machine state *implements* a given abstract system if the following conditions hold:

1. For each $x \; in \mathcal{V}$, the binary $i(x)$ is linked against the `libsel4cp` library and has that library's `main`() function as its entry point.

2. There is an allocated thread control block object in physical memory, which we'll refer to as the Monitor TCB, along with a synchronous IPC endpoint that we'll refer to as the Monitor Endpoint, and a reply object that we'll refer to as the Monitor Reply. The Monitor TCB is not suspended, and has priority 254 and max priority 254. The Monitor TCB has a single `CNode` as its `CSpace`, which contains caps to Monitor Reply in slot 4 and Monitor Endpoint in slot 74. The `VSpace` of the Monitor TCB consists only of the frames required to contain the seL4 Core Platform monitor executable.

3. For each IRQ $i \in I$ there is an IRQ handler capability that allows a thread possessing it to set an endpoint which will be notified of the incoming interrupt i, and to acknowledge received interrupts of the same number i.

4. For each $x \in V$, there is a unique thread control block object in physical memory, which we'll refer to as the x TCB, along with a synchronous IPC endpoint that we'll refer to as the x Endpoint and a unique IPC buffer. The x TCB is not suspended, and has priority $v_p(x)$ and max prioirty $v_p(x)$. There is a unique scheduling context which we'll refer to as the x SC. The x SC has budget and period $v_b(x)$. The Fault EP of the x TCB is the Monitor Endpoint. There is a unique notification object which we'll refer to as the x notification object. If $x \in \mathcal{V}_p$ then there is a unique endpoint object which we'll refer to as the x endpoint. The `CSpace` of the x TCB consists of one `CNode`, which contains the following capabilities:

- An unbadged RW capability to the x notification object in slot 1. We'll refer to this as the x input cap.

- An unbadged capability to the `VSpace` of the x TCB in slot 3.
- An unbadged capability to the x reply object in slot 4. We'll refer to this as the y input cap.
- For each $y \in V$ and $c, d \in C$ such that $\{(x, c), (y, d)\} \in \mathcal{C}$, a RW capability to the y notification object in slot $10 + c$, badged with 2^d.
- For each $y \in V_p$ such that $v_p(y) > v_p(x)$, and $c, d \in C$ such that $\{(x, c), (y, d)\} \in \mathcal{C}$, an RW capability to the y notification object in slot $74 + c$, badged with $2^6 3 + d$.
- For each $i \in I$ and $c \in C$ such that $(x, i, c) \in \mathcal{I}$, an unbadged minted copy of the IRQ handler capability for i, in slot $138 + c$.

The x `VSpace` contains mappings for the frames required to contain the ELF binary $i(x)$. For each mapped memory region $m \in \mathcal{M}$ with $b(m) = x$, a frame is mapped with corresponding physical, virtual addresses, write permission if `W` $\in w(m)$, executable permission if `X` $\in w(m)$, and cached if `C` $\in w(m)$.

2.4 Formal Specification

We provide a partial formalization of the rigorous definition as part of the executable specification of `libsel4cp`. The executable specification is a Haskell that defines data structures for storing platform state and invariants. The Platform Invariants correspond exactly to the abstract systems defined above. Moreover, the relation `relation_cap_map` of `Main.hs` faithfully implements clauses 1,3,4 above.

The verification of the CapDL mapping will require this formalization to be augmented with clause 2, and translated into Isabelle/HOL. Taking on the task of translating the Haskell specification of the mapping into Isabelle/HOL may be a bit of a hassle, but it will not hinder the progress or timeline of the project. After the translation is complete, we will have to decide which version of the specification to stick with moving forward. Given the advantages that come with using Isabelle/HOL for specifying and proving properties, it's likely that we will choose to maintain the Isabelle/HOL version of the specification over the Haskell one, streamlining future development efforts.

Ultimately, the functionality for automatically generating CapDL output based on the user-provided system specifications should target all platforms supported by the seL4 Core Platform. However, seL4CP code is by nature platform-aware (as opposed to platform-agnostic), and the system currently supports only a few development boards. The specification may have to be generalized to be more platform-agnostic as support for new development boards arrives.

2.5 Engineeering and Verification Challenges

2.5.1 Obligations from the `libsel4cp` verification

The `libsel4cp` verifiers assume that the Platform Invariants are preserved by any kernel call in a system that satisfies the platform invariants. This creates a proof obligation, which is best discharged as part of the Isabelle/HOL proofs surrounding verified system initialization.

Moreover, while the seL4 microkernel itself is single-threaded, seL4CP systems run in user mode, which is fully concurrent. This poses a challenge when trying to reason about the

`libsel4cp` library using a Hoare logic precondition/postcondition framework, as the extension of Hoare logic to concurrent programs is a difficult problem to solve. Even if there were established methods for generating verification conditions in such frameworks, the complexity of analyzing preemption and order of execution would exceed the capabilities of our SMT solvers.

To get around this issue, we introduce the concept of thread-local state. We can reason about the parts of the implementation that only involve thread-local state using sequential techniques. However, this creates the need to show that kernel calls made by any thread cannot cause the thread-local state in our projection to cease being thread-local, as part of the proof that the platform invariants are preserved. We are currently investigating whether this can be done in the current or future phases of the project.

2.5.2 Changes required to the initial task

The CapDL initializer acts as the initial task (first user-level thread with access to all resources) for an seL4 system, which initializes the user level in concordance with a user-supplied CapDL specification at startup, then starts all the threads it created and suspends itself.

In contrast with the CapDL initializer (and with the verified CASE initializer), the current version of the `sel4cp-monitor` has a dual purpose: after initializing the seL4 user-level environment to match the user-supplied SDF system specification at startup, it does not suspend itself, but remains active as a fault handler (monitor) for all protection domains.

Consequently, to properly initalize an seL4CP-based system, the CapDL generated by our PoC translator has to launch a “reduced-functionality” `sel4cp-monitor` as a that does not perform the initialization tasks, but can act as the fault handler for the protection domains.

This is a superior design from the security and reliability perspective: the initial process’s `CNode` contains a whole lot of capabilities that need to remain inaccessible, and it’s possible, however unlikely, that an active monitor could be compromised. Moreover, the fact that these caps remain active and accessible makes the proof obligations regarding the preservation of platform invariants much harder to prove. We will implement this change in the future.

2.5.3 Other engineering issues

The build processes for both CapDL loaders integrate tightly with the arcane build system of CAmkES. The CapDL loaders have to be linked against the C output of the CapDL tool, as well as archive containing the ELF binaries that will be loaded: all of this is currently accomplished using `cmake`. Since being agnostic to build systems is an attractive feature of the current seL4CP implementation, disentangling these build processes will be necessary for integration of CapDL loaders into the systems. Preliminary work has been done on this but is not yet completed.