School of Computer Science & Engineering
Trustworthy Systems Group

Verifying the seL4 Core Platform

Verification of the seL4CP implementation
Step 1: High-Level Specification and Approach

Zoltan A. Kocsis

December 2022

1 Summary

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.

seL4CP consists of a small ($\approx$ 280 SLOC) library, `libsel4cp`, which provides abstractions that are easier to use than bare seL4 system calls.

The verification of the seL4 kernel requires the use of interactive theorem provers. Parts of the verification process have to be repeated every time the kernel code changes, and keeping the kernel verified incurs an ongoing cost. For seL4CP we therefore aim for a high degree of proof automation and low proof burden. While SMT solvers can produce “push-button” proofs once set up correctly (with the benefit of automatic re-verification when the implementation changes), getting to that point is still challenging for non-trivial systems, and frequently rely on strong assumptions on the environment of the code to be verified. In contrast we aim to assume no more than the seL4 proofs guarantee.

The goal of this activity is to make seL4 easier to deploy in security-critical scenarios, while raising the assurance levels of both `seL4cp` itself and the future dynamic operating system which will be developed on its base. This activity complements the activity *Verified mapping of seL4CP spec to CapDL*, which aims aims to produce a mapping from the SDF system description format used by seL4CP onto CapDL. Some of the assumptions made in the present verification approach will appear as proof obligations when we verify the seL4CP to CapDl mapping.

2 High-level Specification

2.1 Preamble

2.1.1 Core Platform Abstractions

The seL4 Core Platform is a small operating system and SDK built to run on the proven secure, safe, and reliable seL4 microkernel. The micorkernel itself provides a small number of services for implementing systems, such as capability-based access control primitives, device interrupts, endpoints for message passing, virtual address spaces, threads and scheduling contexts. However, systems built using seL4CP do not use these seL4 services directly: instead, they rely on four simple abstractions provided by the seL4 Core Platform library and monitor.

The four seL4CP abstractions were deliberately designed in such a way that they lend themselves to very efficient implementation in terms of the services provided by the seL4 microkernel, while ruling out common design issues that arise when seL4 abstractions are used directly in certain inappropriate ways.

Protection domains (PDs) constitute the primary abstractions of seL4CP: a PD encapsulates a thread executing a copy of the `libsel4cp` library (linked to a bunch of user-provided code, the actual user program), a virtual address space and capability space in a certain well-defined manner. Each PD also has scheduling information, including a priority, associated with it.

Two protection domains may have a *channel* between them: `libsel4cp` provides functions for sending and receiving asynchronous signals across channels. On the implementation side, the existence of a channel between two protection domains asserts the presence of certain capabilities in the capability spaces of the associated PDs. Receiving and acknowledging *device interrupts* is also implemented using a slight variant of the channel concept.

The seL4 Core Platform provides a mechanism for implementing cross-PD function calls, the *protected procedure call*. The library has functions that allow a PD of lower priority to perform a protected procedure call to a channel with higher priority.

The final abstraction of seL4CP is the *memory region*, which represents a known contiguous range of physical memory. A memory region can be mapped into the virtual address space of (possibly multiple) PDs with various privileges. As with channels, on the implementation side the existence of a mapping between a memory region and a PD asserts the presence of certain capabilities in the capability space of the PD, and facts about the layout of the virtual address space of the PD. These are made precise in the reports on activity *Verified mapping of seL4cp specification to CapDL*, where we give a precise specification of which capabilities should be provided by which abstractions.

2.1.2 System Description Format

An seL4CP system is built from a set of ELF binaries that will run in each protection domain, and an XML *System Description Format* (SDF) configuration file that describes which protection domains will be present in the system, which channels will be present between them, which protected procedure calls will be allowed, etc. The description of the configuration format can be found in the seL4CP manual.[1] These files completely specify the

[1] https://github.com/BreakawayConsulting/sel4cp/blob/main/docs/manual.md#sysdesc

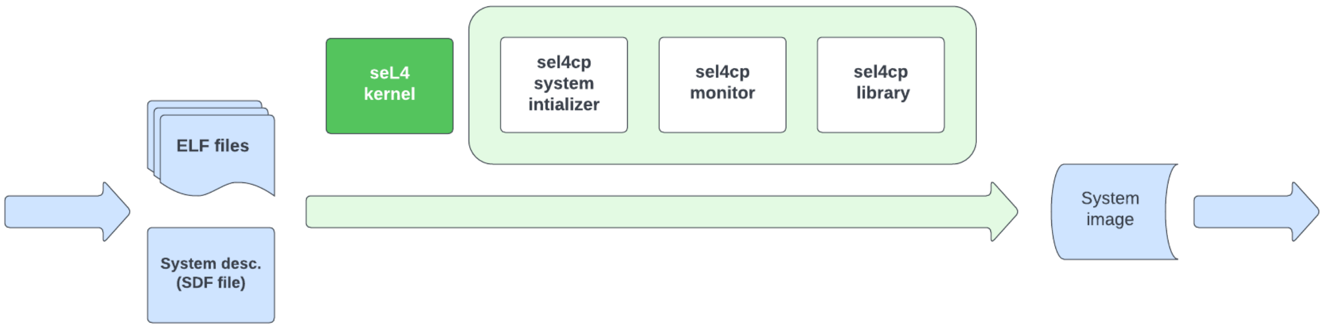

Figure 1: The seL4CP build process. The user provides (blue) ELF files linked against `libsel4cp` that implement the PDs, and an SDF. The seL4CP SDK (green) uses these to build the system image.

system, and are sufficient for the SDK to build the system image, as shown in Figure 1.

Moreover, the seL4 Core Platform has a static structure: properties (such as the number of protection domains, the channels between them, the mappings and permissions of the memory regions, which PDs allow protected procedure calls, etc.) never change during the execution of the system. In the formal model of seL4CP, we will collectively refer to the data specified in the SDF file as the *Platform Invariants*.

2.2 Structure

2.2.1 Model Structure

Verifying the implementation of the seL4 Core Platform library will require several ingredients:

1. A model of the seL4 Core Platform as a state transition system, phrased purely in terms of the Core Platform Abstractions presented above (Platform State).
2. A model of the Kernel/Thread state on the implementation side, phrased in terms of abstractions that the machine and the kernel provide.
3. A relation $\sim$ to tell us which kernel/thread states are consistent with which Platform states.

We can then specify each function of `libsel4cp` by asserting the preconditions for invoking the function, and defining the state transition the function causes in the Platform State. We can then say that a program `f` correctly implements the state transition f precisely if whenever a machine state x is related to the kernel/thread state `X`, the state $f(x)$ of the platform after invoking f, is still related to the state `f`$($`x`$)$ of the kernel obtained by executing `f` in the starting from the state X.

Notice that

1. The Platform Invariants describe the static properties of an seL4CP-based system: the protection domains, channels, protected procedure calls, memory regions, IRQs. These properties can differ between two systems, but remain invariant under state transitions of a single system.
2. The Platform Dynamic State contains the parts of the state that may change with each state transition. This dynamic state includes parts that correspond exactly to some

piece of kernel state, but also *ghost state* which is not, or in some cases cannot be, represented explicitly in the implementation.

The relation $\sim$ can then be specified in two relations as well:

1. A given implementation state satisfies the Platform Invariants. The Platform Invariants only constrain what kernel objects the system will have, and how capabilities to those objects will be distributed. Consequently, the relation between platform invariants and kernel/thread state is completely determined by the SDF-CapDL mapping specified as part of Activity 3.1.1.
2. A given implementation state is consistent with the platform's dynamic state.

Our automated proof will get to *assume* that any and all implementations preserve the Platform Invariants, and will *show*, using this assumption, that the implementations preserve the dynamic state relation as well. The assumption of Platform Invariant preservation is safe: it is a proof obligation for verified system initialization that the verified system initializer will initialize the platform into a state consistent with the Platform Invariants. From there, it should be possible[2] to show that no kernel call made by any protection domain will ever put the system into a state not consistent with the Platform Invariants. It follows that no implementation can fail to preserve the Platform Invariants.

2.2.2 Oracles and thread-local state

While the seL4 Core Platform is an operating system, it runs in user mode on top of the seL4 microkernel. The kernel itself is single-threaded, but user mode is fully concurrent. In principle, it would be possible to develop a Platform State structure that represents the entire static and dynamic state of an seL4CP-based system, this would be both overkill for our present purposes (since we only intend to prove properties about single instances of `libsel4cp`, and each PD runs in a single thread), and would make automated reasoning difficult or impossible. We wish to reason about our code in a Hoare-style precondition/postcondition framework, and extension of Hoare logic to concurrent programs is not an entirely solved problem: even if there were tried-and-true methods for verification condition generation in such frameworks, the combinatorial explosion caused by having to analyze preemption, order of execution, etc. would exceed the capabilities of our SMT solvers.

Consequently, our formal construction of the Platform Dynamic State will not include the whole state of the platform, only the state pertaining to the current protection domain. Similarly, the kernel state will not include the whole state of the system, only the state that is relevant to the implementation of `libsel4cp` inside the currently running thread.

To make this precise, we introduce the notion of *thread-local state*. Consider the abstract specification of seL4 and its global kernel state S. We say that a `PSpace`-datum d in the abstract kernel state S is part of the *thread-local state* of a thread t in S if the following holds: in any global kernel state reachable from S, the datum d cannot change unless the thread t makes a kernel call. In particular, every sequence of kernel calls that changes d includes at least one call made by the thread t.

The kernel state of a thread t is required to be a projection (subset) of the state of the entire system (analogous to e.g. a value of type `KernelState` in `SEL4.Model.StateData` of `l4v`)

[2]This requires the decoupling the initializer functionalities from the monitor.

consisting only of thread-local state of t. Since the thread-local state can only change when t is executing, we can reason about the parts of the implementation that involve only the thread-local state using purely sequential techniques. Naturally, this creates an obligation for us to show that none of the kernel calls made by t cause the thread-local state to cease being thread-local: we eventually aim to verify 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.

Notification delivery and IPC depend fundamentally on concurrency, in the sense that the received notifications and procedure calls depend on what the other protection domains chose to do. We can model the concurrency-dependent operations using *oracles*: variables whose values tell us what the concurrency-dependent calls *will* return in the future, when the current thread chooses to make a concurrency-dependent call.

Normally, such oracles could be represented as lists or more advanced data structures, but since our verifier encodes properties in SMT for automated verification, we had to choose a representation that avoids unbounded data structures and recursion. We opted to model the oracles as single-use. At the beginning of each execution iteration, the oracles become available. The availability of the corresponding oracle becomes a precondition of each kernel call, so a kernel call cannot be made unless the oracle is available. Similarly, the unavailability of the corresponding oracle becomes a postcondition, so the oracle becomes invalidated/unavailable after a kernel call which might have affected the return value of the next call.

2.2.3 Specification

The full implementation state of a thread and the associated protection domain consists of the following:

- The Platform Invariants,
- the dynamic platform state P of the protection domain, including the states of the platform oracles,
- the thread-local kernel state K of the thread, and
- the states of the kernel call oracles.

The **seL4 Core Platform specification** consists of

- pre- and postconditions for the platform state transitions, phrased in terms of the platform invariants and the dynamic platform state,
- pre- and postconditions for the kernel state transitions, phrased in terms of the full implementation states (both platform and kernel states).

A C implementation `F` of a platform state transition f in terms of kernel state transitions is considered correct precisely if executing `F` on a related pair $P \sim K$ of platform and kernel states where P satisfies the preconditions for the platform state transition f results in a related pair $\texttt{F}(P) \sim \texttt{F}(K)$ where $\texttt{F}(P)$ satisfies the postconditions for the platform state transition f. The goal of our SMT-based verifier is to establish that the implementations of the `libsel4cp` handler loop and all user-facing `libsel4cp` calls are correct.

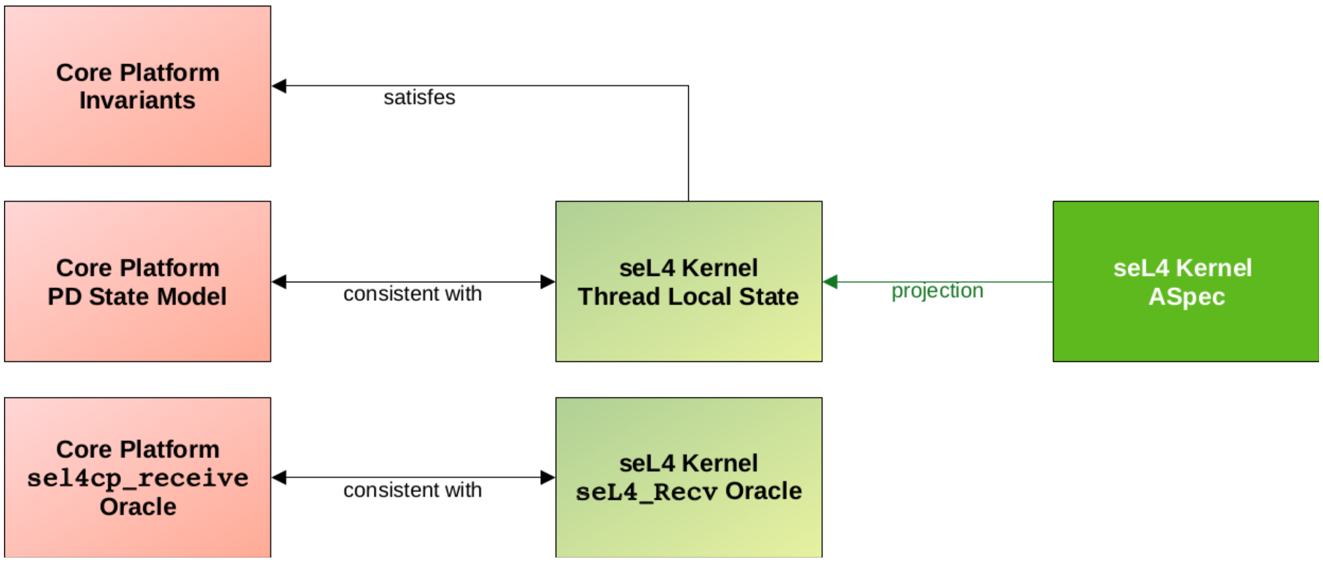
Core Platform Invariants
satisfes
Core Platform PD State Model
consistent with
seL4 Kernel Thread Local State
projection
seL4 Kernel ASpec
Core Platform sel4cp_receive Oracle
consistent with
seL4 Kernel seL4_Recv Oracle

Figure 2: Overview of the preserved relation between the Platform (red) and Kernel (light green) contexts, and the relationship between the Thread Local State and the full abstract kernel state (dark green).

2.3 Formal Specification

An executable specification of `libsel4cp` is delivered as a Haskell program. The program defines data structures for storing the platform invariants and the dynamic platform, kernel and oracle states, and defines pre- and postconditions for the state transitions (usually in weakest-precondition form). The specification is available in the `july/spec` directory of the Github repository. A hyperlinked HTML export of the comments and documentation is included.

The Haskell program defines the data structures in a bounded fashion, which ensures that they will be faithfully representable in `QF_ABV` theory that our SMT solver use. The model of thread-local kernel state included in the code, while simplistic, will be sufficient to develop and test our initial SMT-based verifier. The model will be expanded continuously as verifier development progresses. However, porting the entire kernel model from `SEL4.Model.StateData` would take considerable time, and will not be attempted until the feasibility of SMT-based verification is demonstrated successfully in the present phase.

Engineering challenges

The use of Haskell in the formal specification allowed for efficient initial prototyping and development. Since that the final implementation will utilize an SMT-based verifier, and related properties will need to be proven using Isabelle/HOL, it will be necessary to manually translate the Haskell specification into both Isabelle/HOL and Python. Overall, while the translation work may be somewhat inconvenient, it will not impact the success and timelines of the project.

However, once the Haskell specification has been translated into Isabelle/HOL and SMTLIB2, we will have to decide which versions of the specification to maintain, and use going forward. Given the benefits of using Isabelle/HOL for specifying and proving properties, it is likely that the Isabelle/HOL version of the specification will be the one that is maintained over the Haskell one, which will simplify future development efforts.

3 Verification Approach

3.1 C Semantics

Since the library and monitor of the seL4 Core Platform are largely implemented in C, their verification requires a formal semantics for C that is able to deal with C's model of pointers and heap memory correctly and without oversimplifications.

After deliberation, we opted to use the same `c-parser` tool that is used to create the Isabelle/HOL semantic model of the C source code of the seL4 kernel. This tool induces a semantics for C by translating the C source into Schirmer's SIMPL [Schirmer, 2008] programming language, which has a well-defined operational semantics in Isabelle/HOL [Schirmer, 2006]. This choice allows us to use the exact same semantics that the kernel verification used. Moreover, SIMPL can be further translated (exported) into the SydTV-GL graph language in a semantics-preserving manner. The SydTV-GL graph language is already used as the substrate of the seL4 kernel's translation validation [Sewell et al., 2013] and worst-case execution time analysis tools.

The `c-parser` handles a reasonable subset of the C99 standard, and accounts for architecture-defined behaviors such as endianness, or the number of bits in an `int`. The most important limitations of the subset of C implemented by the tool are the following:

1. No `goto` statements.
2. No fall-through cases in `switch` statements.
3. No `union` types.
4. No taking the address of local variables.

With the exception of one instance of number 4, these operations are already absent from the current seL4CP implementation, and we do not expect adherence to these restrictions to become a constraint or difficulty in the future development of the seL4 Core Platform.

3.2 Verification conditions from SydTV-GL

To verify the implementation of the seL4 Core Platform, we have to transform each function into a logical verification condition which can be handed to an SMT solver. This verification condition should be the most general precondition relative to the function's specification: if our SMT solvers can confirm that the negation of the verification condition is unsatisfiable, we can consider the function correctly implemented.

The `libsel4cp` and `monitor` source code is first put through the `c-parser` tool described above, then exported (using `SimplExport`) to the SydTV-GL graph language, the same language used for the seL4 kernel's translation validation and worst-case execution time analyses, which also use SMT solvers. The resulting SydTV-GL code is an unstructured graph program representing a control flow graph, whose semantics refines the C semantics induced by the `c-parser`, and which consists of three sorts of nodes:

- `Basic` nodes which update local variables;
- `Cond` nodes which perform conditional jumps; and
- `Call` nodes which perform calls to other funtions.

With a few modifications, the algorithm of Barnett and Leino [Barnett and Leino, 2005] (from here on BRL) can be adopted for computing the weakest precondition of SydTV-GL functions, while remaining sound in the presence of loops. The method involves transforming the graph language program into a reducible control-flow graph in dynamic (as opposed to static) single-assignment form. While SMT performance is difficult to predict in advance, the dynamic single-assignment transformation should result in logical expressions that provide enough slack for SMT solvers to attempt non-trivial proofs. We plan to implement the BRL algorithm in Python, reusing parts of the already-existing Python tools that process SydTV-GL.

Once verification condition generation is implemented, we can verify implementation correctness function-by-function. We assume that we can treat the targets of each `Call` node as black boxes fully specified by their corresponding theorems. Once all the function-by-function proofs are completed, a topological sort of the call graph is produced to show that the resulting correctness proof is non-circular. Whenever possible, loop invariants will be derived automatically using simple SMT-guided heuristics.

As stated above, a C implementation `F` of a platform state transition f in terms of kernel state transitions is considered correct with respect to its specification precisely if executing `F` on a related pair $P \sim K$ of platform and kernel states where P satisfies the preconditions for the platform state transition f results in a related pair $\mathtt{F}(P) \sim \mathtt{F}(K)$ where $\mathtt{F}(P)$ satisfies the postconditions for the platform state transition f. That is, the verification conditions will have to account for satisfaction of the pre- and postconditions, and establish that the implementation relation $\sim$ between the platform and kernel states is preserved.

3.3 Proof Construction

1. Each C function in the implementation is annotated with its formalized specification.
2. The C source code is translated into SIMPL using the `c-parser`.
3. The SIMPL is translated into SydTV-GL graph language code via `SimplExport`.
4. Verification conditions are generated by the Python implementation of the BRL algorithm and the invariant-finding heuristics.
5. The verification conditions are checked by multiple SMT solvers.

3.4 Proof Aims

We aim to prove the following properties about the Core Platform implementation:

1. Platform code in `libsel4cp` and the `monitor` does not fail, and no undefined behavior is encountered according to the C semantics induced by the `c-parser`. Among other things, this means that assertions never fail, no null pointer dereferences or out-of-bounds accesses are performed outside of user code.
2. All user-facing `libsel4cp` calls terminate, even calls with misinformed input.
3. The handler loop satisfies its specification. The handler loop never terminates. Once an iteration of the loop is completed, `libsel4cp` code will have either received all pending channel notifications, and invoked the corresponding channel's user-supplied `notified` method exactly once, or handled a protected procedure call.

3.5 Limitations, Mitigation

1. Our verification conditions are handled by SMT solvers. In an ideal world, if an SMT solver states that a given set of formulae is satisfiable, then we would know (have a proof) that the formula is indeed satisfiable, and vice versa if it states that the formula is unsatisfiable, then we would know that the formula is UNSAT. Alas, none of the state-of-the-art solvers are formally verified to be correct, and while it is possible to replay/produce proof certificates (which can then be independently verified), that is very time consuming.

To deal with this threat to validity, we adopt mitigation techniques similar to those already adopted by the kernel's translation validation: we emit all SMT queries in the standard SMTLIB2 format in the quantifier-free theory of arrays and bitvectors, and allow the user to plug in various implementations of SMTLIB2-compatible solvers. We reuse the parallel solver mechanism (and the in-development continuous integration framework) from translation validation.

2. We assume sequential execution semantics throughout. This assumption is reasonable for the seL4 kernel, but in contrast with the kernel, the Core Platform runs in user mode. Strictly speaking, full correctness would require a much more detailed semantics accounting for the fact that the current thread may be preempted, registers may be restored (with an unverified kernel potentially incorrectly!), etc. While we believe that, using the guarantees proved about the seL4 kernel, it should be possible to prove a *meta-theorem* showing that this assumption is safe (i.e. every proof of sequential correctness can be expanded mechanically into a proof of full correctness), a rigorous treatment or formalization of such possibilities is strictly outside the scope of the current work.

4 Roadmap

The current deliverable included a high-level specification of seL4CP showing what kinds of functional properties should be established about the implementation, along with a verification plan explaining how SMT solvers will be used to show implementation correctness.

The implementation of the verifier will proceed as follows:

1. First, we create the infrastructure required to export the `libsel4cp` and `monitor` source code into SydTV-GL graph language. This involves non-trivial preprocessing and stubbing of `libsel4` calls. Options to automate this process in a way that's resilien to seL4 Core Platform implementation changes will be explored at a a later stage.
2. We fork the translation validation tool `graph-refine` and implement checks, graph transformations and heuristics required for the BRL algorithm.
3. We use the BRL algorithm to prove the absence of *undefined behavior* in `libsel4cp`. This does not require the formalized specification of the library, and will allow us to confirm the soundness and performance of our verification method before the formalized SDF semantics is finished.
4. We progress towards automated proofs of functional correctness of `libsel4cp` and `monitor`.

References

Mike Barnett and Rustan Leino. Weakest-precondition of unstructured programs. In *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 82–87. ACM Press, September 2005. ISBN 1-59593-239-9. URL https://www.microsoft.com/en-us/research/publication/weakest-precondition-of-unstructured-programs/. 8

Norbert Schirmer. Verification of sequential imperative programs in isabelle-hol. 2006. TU Munich examined thesis, PhD. 7

Norbert Schirmer. A sequential imperative programming language syntax, semantics, hoare logics and verification environment. *Archive of Formal Proofs*, February 2008. ISSN 2150-914x. 7

Thomas Sewell, Magnus Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 471–481, Seattle, Washington, USA, June 2013. ACM. 7