School of Computer Science & Engineering
Trustworthy Systems Group

Verifying the seL4 Core Platform

Verification of the seL4CP implementation Step 2: Initial Verifier

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.

The seL4CP implementation 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 proof development 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 verification with a high degree of proof automation and low proof burden. While SMT solvers 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 a challenge for non-trivial systems, and frequently relies 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.

2 Implementation Progress

2.1 Roadmap

The development of the verifier follows the previously indicated approach:

1. First, we create the infrastructure required to export the `libsel4cp` and `monitor` source code into SydTV-GL graph language.
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`.

2.2 Infrastructure tools

2.2.1 C parser

We decided to use the same tool, the `c-parser`, that is used to create the Isabelle/HOL semantic model of the seL4 microkernel. This tool translates the C source code into Schirmer's SIMPL [Schirmer, 2008] programming language, which has a well-defined operational semantics in Isabelle/HOL. This allows us to use the same semantics as the kernel verification. Additionally, SIMPL can be translated into the SydTV-GL graph language using the `SimplExport` tools, in a way that preserves the semantics.

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.

Most of 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.

However, the source code of the `handler_loop` in `libsel4cp` did contain one instance of the fourth limitation, where the address of the local variable `badge` had to be taken as a result of an odd design decision in `libseL4`. Fortunately, a simple workaround was available by introducing an auxiliary global to store this badge (required changes: 4 LOC).

Due to the fact that the seL4 Core Platform library and monitor are implemented in terms of other libraries such `libseL4`, which were not written with the `c-parser` in mind, we had to write preprocessing scripts before we could read the `libsel4cp` and `monitor` using the parser and successfully dump it into graph language using `SimplExport`. Options to automate this process in a way that's resilient to seL4 Core Platform implementation changes will be explored at a later stage.

Statistics: with all includes, exporting `libsel4cp` yields a graph consisting of 5627 nodes, divided into 320 functions. However 302 of these come from `libseL4` and other includes, and

only 6 of these are relevant to the verification, leaving a total of 18 functions (about 110 nodes) to be verified. The main handler loop consists of 42 nodes.

The corresponding numbers are slightly larger for the monitor, which has 24 functions to be verified, with a total of 648 relevant nodes. Note that this latter number includes the initializer, and will decrease substantially once the initializer and monitor functionalities are decoupled.

Exports of the monitor and libsel4cp can be found in the `nov/exports` directory of the Github repository.

2.2.2 Graph language

The SydTV-GL graph language is already used in the seL4 kernel's translation validation and worst-case execution time analysis tools [Sewell et al., 2013]. We started verifier development by forking the translation validation toolchain. By using the toolchain, we were able to ensure that our implementation would be consistent with the existing developments and allowed us to reuse a large chunk of SydTV-GL functionality for our purposes. We improved the type-safety of the toolchain by adding new interfaces with additional checks and constraints, which helps to prevent programming errors and improves their overall trustworthiness. The translation validations tools are undergoing maintenance and development as part of the *Binary Correctness* project. Once the current phase of these projects concludes, we will look into merging these improvements back upstream so that they can be incorporated into the main version of the toolchain.

2.3 Verifier Implementation

2.3.1 Graph language processing

The graph language processing is an improved version of the one inherited from translation validation: after our changes, it is complete, and able to read and process the exported `libsel4cp` and `monitor.c` files.

2.3.2 BRL algorithm implementation

We are using a minor variant of the "weakest precondition for unstructured programs" algorithm of Barrett and Leino [Barnett and Leino, 2005] to figure out the weakest conditions that need to be met in our control flow graphs to prove correctness.

We first apply a loop elimination transformation to the SydTV-GL graph, resulting in a control-flow graph that is free from cycles, and that is correct only if the original program was correct. The next step involves transforming the program into a control-flow graph in dynamic single-assignment form. This change makes it easier for SMT solvers to do their job. We have implemented loop elimination and the dynamic single-assignment transform, along with a comprehensive test suite.

Finally, the weakest precondition computation is applied to the acyclic single-assignment control flow graph to generate the verification condition which is handed to the solvers. The verification condition can then be handed to the SMT solver. The verification condition generator is still under development, so *the verifier is not yet able to verify the functional correctness of* `libsel4cp`.

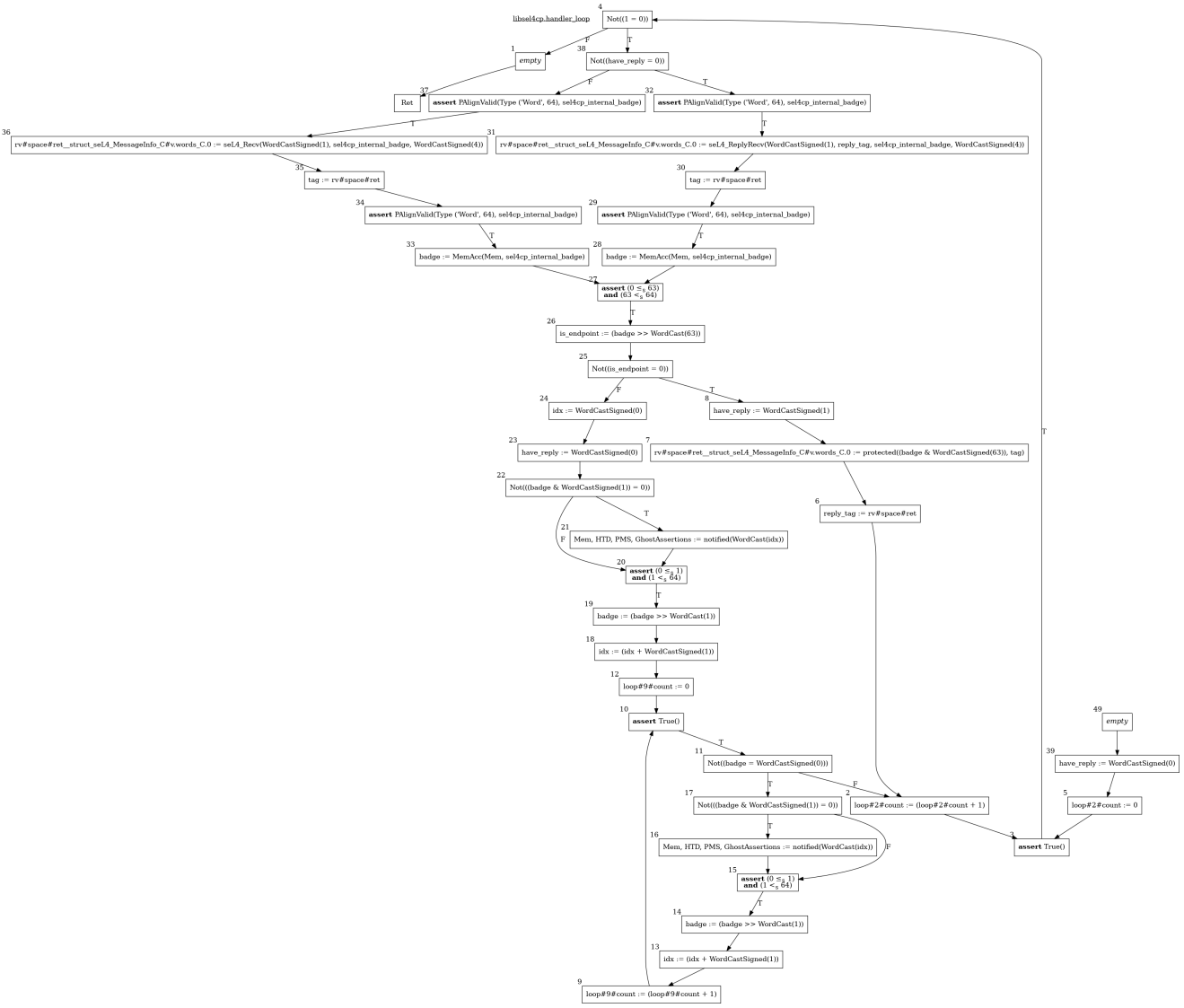

Figure 1: The handler loop of `libsel4cp` can now be exported to the SydTV-GL graph language, yielding the control-flow graph above. The false branches of assert statements lead to the `Err` node (omitted).

We will first use BRL algorithm to prove the absence of *undefined behavior* in the library. This does not yet require the formalized specification of the library to be imported, and will allow us to confirm the soundness and performance of our verification method before the full SMT translation of the specification is finished. We expect to reach this milestone before the end of the year. From there, we expect rapid progress towards proofs of functional correctness.

2.3.3 Monitor-initializer decoupling

The seL4CP monitor is a C program that is used to initialize the user-level environment of the kernel to match the user's specifications. This includes starting threads, distributing capabilities and loading ELF programs from the user's image file. Once the system is initialized, the monitor remains active and monitors all protection domains for faults.

Due to the development of verified system initialization (Activity 3.1.1.), the system initializer functionality of the monitor will be taken over by the CapDL initializer and eventually the verified CASE initializer. Unlike the monitor, the CASE initializer shuts down (suspends its own thread) after initialization. This is a much better design from the security and reliability perspective: the initial process's `CNode` contains many capabilities that need to remain inaccessible, and it is possible, however unlikely, that an active monitor could be compromised.

The system initializer functionality of the monitor will become redundant with a verified initializer, but the fault handler functionality will still be performed by the seL4 Core Platform's monitor. This requires us to produce a version of the monitor that's decoupled from the initializer. The requirements and specifications for the new version are to be determined and then implemented. This decoupling is a prerequisite for both the functional correctness verification of the monitor (as we won't verify the redundant initializer parts) and the verification tasks that form part of the SDF-CapDL mapping.

Engineering challenges

We have produced an initial implementation, which is now able to read and process the SydTV-GL export of the seL4 Core Platform, but we have not yet reached the stage where it can generate the the verification conditions required to show functional correctness of `libsel4cp`. We anticipate reaching this by the end of the year.

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/. 3

Norbert Schirmer. A sequential imperative programming language syntax, semantics, hoare logics and verification environment. *Archive of Formal Proofs*, February 2008. ISSN 2150-914x. 2

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. 3