First steps in verifying the seL4 Core Platform

Mathieu Paturel, Isitha Subasinghe, Gernot Heiser
gernot@unsw.edu.au
Trustworthy Systems Research Group, UNSW Sydney

ABSTRACT

We report on our initial effort to formally verify the seL4 Core Platform, an OS framework for the verified seL4 microkernel. This includes a formal specification of the seL4 Core Platform library, an automated proof of its functional correctness, and a verified mapping of the seL4 Core Platform's System Description to the CapDL formalism that describes seL4 access rights and enables verified system initialisation.

ACM Reference Format:
Mathieu Paturel, Isitha Subasinghe, Gernot Heiser. 2023. First steps in verifying the seL4 Core Platform. In *14th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '23), August 24–25, 2023, Seoul, Republic of Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3609510.3609821

1 INTRODUCTION

Formal verification uses mathematical methods to confirm that software meets its predetermined specifications; it is one of the few techniques that can positively establish the absence of bugs in software.

The correct operation of a system depends on the underlying hardware, the operating system (OS) kernel, and higher-level frameworks that provide the environment in which the system executes. Consequently, bugs in the kernel or the frameworks can and will compromise the system's correctness and overall reliability.

The high-performance **seL4** microkernel was the first ever general-purpose OS kernel with a formal proof of implementation correctness, which was later extended to the binary code (taking the compiler out of the trust chain), proofs of security enforcement, and proofs of its worst-case execution time [Klein et al. 2014]. Presently, the full proof chain exists for the 32-bit Arm and 64-bit RISC-V architectures, with the implementation-correctness proof also available for the 64-bit x86 architecture. At the same time, seL4 demonstrates best-in-class performance [Mi et al. 2019], making it the ideal foundation for secure and dependable real-world systems. The seL4 microkernel's ongoing influence of seL4 was recently recognised by the 2022 ACM Software System Award.

However, far from a full OS, seL4 is a microkernel which provides only basic mechanisms for securely multiplexing hardware. Its API is consequently policy-free and low-level, making development of performant and correct systems on top costly and requiring a high level of expertise, and generally creating a high barrier to uptake.

The recently developed *seL4 Core Platform* (seL4CP) [Heiser et al. 2022] addresses this challenge by providing a small set of higher level abstractions that are easy to use for building modular, yet performant, systems that leverage seL4's isolation properties [Parker 2022]. It also comes with an SDK that simplifies system generation.

The seL4CP is still not an OS, but a framework for building OS services and applications. It achieves much of its simplicity by restricting the application domain: Instead of striving for generality (as seL4 does), the seL4CP is designed for systems with a static architecture, i.e. one where all components and their interactions (but not necessarily the implementation of those components) are known at system-configuration time. These restrictions, while incompatible with desktop or cloud hosting environments, are sufficient to support most IoT and cyber-physical systems.

Along with a leap in usability, the simplicity of the seL4CP has another advantage: module implementations tend to be very simple. Combined with the fact that module interfaces are enforced by verified seL4 mechanisms, this should dramatically simplify the task of verifying seL4CP-based systems. However, the full benefit of verifying such systems will only be realised if the seL4CP itself is verified.

Simplicity helps here as well, as the implementation of the seL4CP itself is also fairly straightforward, simple enough to experiment with a more *automated verification process*.

Verification of the seL4 kernel used *interactive theorem proving* in Isabelle/HOL, which allows the construction of elaborate, machine-checked proofs. While powerful in what it can prove, this manual approach is highly labour-intensive. Furthermore, it requires manual re-verification whenever the code changes. This is in contrast to *SMT solvers*, which are tools that, once set up, can verify functional properties fully automatically, known as "push-button verification". This approach was deployed in the binary verification of

APSys '23, August 24–25, 2023, Seoul, Republic of Korea

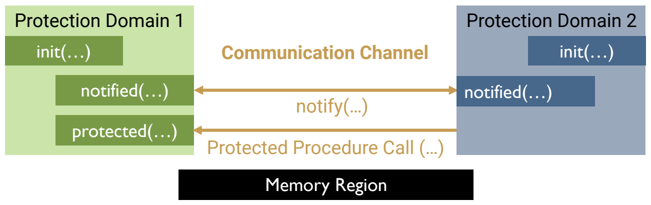

Figure 1: seL4 Core Platform abstractions at a glance.

seL4 [Sewell et al. 2013], and more recently in functional-correctness proofs of simple operating systems [Nelson et al. 2019, 2017].

Significant challenges remain. The seL4 proofs could assume strictly sequential execution, thanks to the non-preemptible implementation of seL4 [Peters et al. 2015]. This assumption does not hold for code that executes in user mode, such as the seL4CP and system components built on top, as their execution can be preempted at any time. Making verification tractable requires taming this concurrency (see last paragraph of Section 2). Previous push-button verification approaches assume these challenges away, which drastically limits the guarantees that can be obtained from verification.

Here we report on the first successful steps in the formal verification process for the seL4 Core Platform. Our contributions are:

(1) A formal specification of the seL4CP-seL4 interface library (`libsel4cp`);
(2) An automated functional correctness proof, showing that the `libsel4cp` implementation satisfies this specification.
(3) A verified mapping of the seL4CP system specification to the CapDL formalism [Kuz et al. 2010] which describes access rights in seL4 and enables a verified system initialisation.

2 SEL4 CORE PLATFORM OVERVIEW

The seL4 Core Platform implementation consists of two components, which are linked to user-provided module implementations by the seL4CP SDK:

- The interface library `libsel4cp`, which maps seL4CP APIs to seL4 system calls. It implements an event loop that invokes user-provided handler functions. The total size of `libsel4cp` is presently only about 280 SLOC of C;
- `sel4cpinit`, a boot task which allocates system resources. The design for static architectures makes it possible to restrict this allocation to system startup time. After startup, `sel4cpinit` remains active as a fault handler.

The programming model presented by the seL4CP API is extremely simple, all abstractions are summarised in Figure 1:

- The **protection domain** (**PD**) is the process abstraction; it provides a simple, event-driven execution model, where the user provides the implementation of a handler, called the `notified` function. The user must also provide an implementation of the `init` function, which is called by the system exactly once at startup time.
- A **communication channel** (**CC**) connects exactly two PDs. Communication is asynchronous: a PD may `notify` a CC, which will invoke the `notified` function of the PD at the other end of the channel, passing the identity of the notifier.
- A **memory region** (**MR**) may be mapped into one or more PDs, with potentially different access rights, providing shared memory for communication.
- A **protected procedure call** (**PPC**) is an operation on a CC that invokes the `protected` function in the PD across the channel, which executes synchronously to the caller. The `protected` entrypoint is optional, meaning only some PDs, informally called "servers", may be invoked via a PPC – the PPC is the seL4CP's equivalent of a system call in a monolithic OS. The system requires (but does not enforce programmatically) that a PPC can only go to a higher priority PD (thus preventing deadlock).

An seL4CP-based system is then a collection of concurrently executing PDs (modules) that communicate via shared memory, synchronise via notifications sent along channels, and invoke servers via PPCs. The composition is defined in a formalism called the *system description format* (SDF).

PDs are single-threaded. (Support for multi-threaded applications is on the roadmap. It will allow multiple PDs, running on different cores, to share an address space, i.e. code and data [Trustworthy Systems 2023].) Concurrency within a PD is tamed; the seL4CP guarantees that the `init`, `notified` and `protected` functions execute atomically with respect to each other, eliminating the need for any concurrency control inside a PD, in the spirit of the CSP model of Hoare [1978].

3 GLOBAL CORRECTNESS

Ultimately our goal is to prove the correctness of an seL4CP-based system. This requires specifying the system in terms of a *global* state machine, which holds the externally visible state of the PDs, and a *trace* (temporally ordered list of API calls and shared memory writes made by the PDs, which act as state transitions for this state machine). Correctness can then be defined in terms of permissible traces.

This requires *guarantees* provided by the seL4CP, consisting of formally verified *theorems* about the traces possible in *any* system implemented in terms of the fundamental seL4CP abstractions. These guarantees require a proof that the implementation of the `libsel4cp` and `sel4cpinit` libraries are correct, based on the formal specification of seL4 itself.

A representative guarantee is:

> (†) If a PD p sent a notification on a channel c, whose other end is PD q, using the `sel4cp_notify(c)` API call, then eventually (but no sooner than the next time PD q makes a call to receive notifications) the PD q will execute the `notified(c)` function.

This long-term goal requires overcoming significant challenges:

1. The guarantees required of the seL4CP combine complicated *liveness* (finiteness of the trace) and *safety* properties (ruling out certain finite prefixes of the trace). Verification of such properties is hard with interactive theorem proving and out-of-reach for effective automated techniques; the common theories implemented in SMT solvers do not allow reasoning about unbounded traces.
2. One has to show that the semantics captures the whole external state that a given protection domain may observe.
3. Property † can only hold subject to certain scheduling restrictions. For example, if a protection domain r with a priority higher than p monopolises q with PPCs, the notification will never be processed. This is a legal scenario, which can be prevented by limiting r's time budget [Lyons et al. 2018], but requires reasoning about scheduling behaviour which seL4's current abstract specification leaves undefined.

4 LOCAL CORRECTNESS

As a first step we start verification at a local level. Crucially, we carefully express the local correctness in such a way that it can be reasoned about using SMT solvers.

4.1 Approach

We specify the seL4CP API (as implemented in the `libsel4cp` library) in terms of a *local* state machine, which contains only the state pertaining to the code executing in a single PD, and describes:

1. the execution state of the PD making the API call,
2. the SDF-specified static configuration of the system, and
3. the observations that one can make during the current execution about the state of the rest of the system (such as receive calls, shared memory accesses), modeled as single-use oracles.

Intuitively, single-use oracles are *filled* by the global state machine, and *consumed* by the local one. This allows the former to summarise and communicate complex information to the latter (notably, information only available in the traces).[1] For example, when a PD makes a receive call, it consumes the receive oracle and deduces the return value from it. Beforehand, the global state machine filled the oracle appropriately by observing the trace of the system as a whole.

Our formal specification of `libsel4cp` describes how the various seL4CP API calls made by the currently executing PD should affect this local state machine. In terms of the static configuration of the system, and the single-use oracles, we are able to state guarantees that a correct seL4CP implementation (and by extension, every piece of user code running inside a PD) should provide. For example, we state that a correct implementation will not make a `sel4cp_notify(c)` call to a non-existent channel c, nor will it make a PPC to a PD that has lower or equal scheduling priority than the currently executing PD – we are able to guarantee these on verified systems even though the current implementation of the library does not programmatically enforce the restrictions. The correctness condition for the main `libsel4cp` handler loop, which executes on every PD, states:

> (⋆) The handler loop never terminates. It will make a call to receive notifications and PPCs exactly once per iteration, and will correctly handle any and all responses returned by the receive call (according to the single-use oracle), including calling `notified(c)` if a notification was received on a channel c, and `protected(c, m)` if a PPC was received on channel c with argument m. Furthermore, it will not make "phantom" calls such as calling `notified(x)` on a channel x on which no notification was received.

The local correctness conditions were derived with the global correctness in mind, ie. we expect to use them in a global proof specified in terms of traces. For example, one would appeal to the local condition ⋆, along with a delivery guarantee coming from the kernel to establish the condition †. We emphasise that these local conditions are necessary but that by themselves not sufficient to prove conditions like †. Establishing global conditions will require some additional, manual proof work in interactive theorem provers.

4.2 Formalised specification

We write the initial specification in a constrained subset of the Haskell programming language. The local state machine is defined as an algebraic data type, and each of the seL4CP

[1]From the perspective of the local state machine, oracles predict the future, hence the name.

APIs is defined in the form of a weakest precondition: for each API call `f()` and property φ we describe the weakest condition $F(\varphi)$ under which the call `f()` either does not terminate, or it terminates and upon termination successfully establishes the condition φ. The correctness guarantees are also specified: e.g. the handler loop iteration gets annotated with explicit pre- and postconditions requiring that property $\star$ holds.

4.3 Implementation Relation

We can now verify the `libsel4cp` implementation against this local specification. This requires a model of the seL4 kernel state, and an implementation relation that relates a momentary state k of the underlying seL4 kernel to the momentary state h of the local seL4CP state machine. This *implementation* relation $h \sim k$ holds true if, and only if, the current seL4 kernel state accurately implements the local seL4CP state.

To define this relation inside the Haskell specification, we also need a state machine describing (a part of) the seL4 kernel state, and an axiomatisation of how seL4 kernel calls change this state. Fortunately, there is already an executable model of the seL4 kernel written in Haskell, which is an important intermediate artefact in the seL4 correctness proof [Klein et al. 2014], and is therefore kept in sync with kernel changes (although it is not verified). The kernel state we use as the domain of the implementation relation is a manually derived (i.e. presently unverified) small projection (restriction to a tiny subset) of the executable Haskell model. Similarly to the seL4CP API calls, the seL4 kernel calls are also axiomatised in terms of weakest preconditions. These do not need to be full specifications of all possible kernel calls: they only specify the effects of those calls that we know will occur while executing the `libsel4cp` implementation, and forbid all others.

In seL4 all system objects and resources are accessed through capabilities [Dennis and Van Horn 1966]. seL4 capabilities are part of kernel state and are referenced at user-level by references (*CSpace indices*).

Recall that the static configuration of the system (i.e. the SDF spec used to generate it) is one of the constituent parts of the local state machine for the seL4CP. The implementation relation relates the static configuration of the system to the capability distribution in the corresponding implementation kernel state. This relation is defined in a way such that exactly one capability distribution corresponds to a valid implementation of a SDF spec. In Section 5.2 we describe how knowing this capability distribution allows us to implement formally verified system initialisation for seL4CP systems.

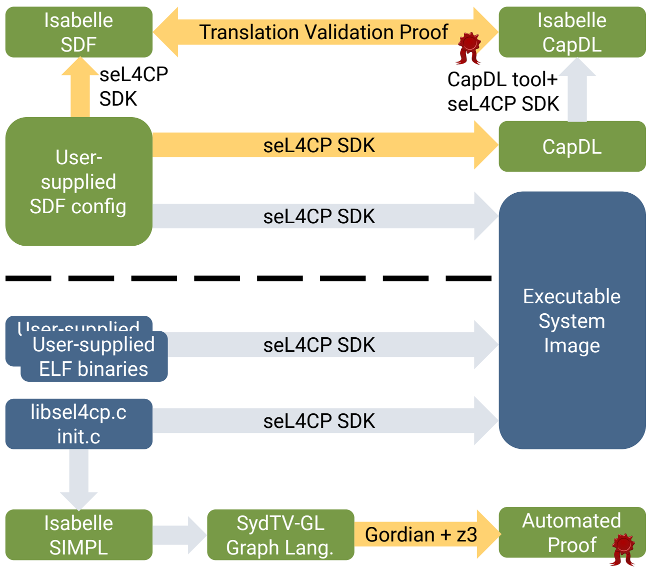

Figure 2: The full verification pipeline for the automated verification of the seL4 Core Platform library and the translation validation of the CapDL export. Formal artefacts are green, informal ones blue. Yellow arrows indicate new translation and verification tools developed over the course of the project, light grey arrows are the result of prior work. The dashed line separates the two sets of proofs (CapDL generation and `libsel4cp` implementation).

Once the implementation relation is defined, the local correctness of the implementation can be established by showing that:

(1) the `libsel4cp` implementation maintains the implementation relation (simulation-proof like), i.e. if $h \sim k$ holds for some platform state h and corresponding kernel state k, and making an seL4CP API call `f()` will leave the platform in some new state h', then executing the implementation of `f()` starting from a kernel state k leaves the kernel in some related state k' (i.e. for all h, k, if $h \sim k$ holds then $h' \sim k'$ holds);
(2) the implementation upholds all the correctness guarantees (e.g. that the main handler loop provides the $\star$ guarantee defined above).

The final formalised specification consists of 812 lines of Haskell code. The use of the constrained subset of Haskell is meant to ensure that the specification can be faithfully rendered in the very limited specification language used by the SMT solvers in the process of automated verification.

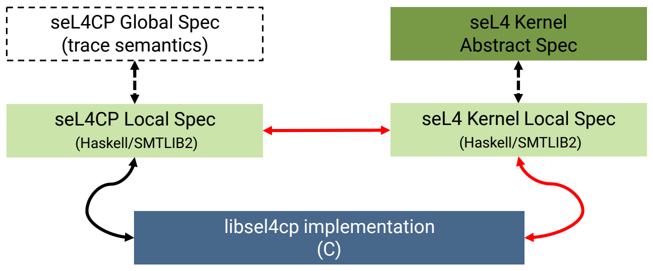

Figure 3: Relationships between specifications for `libsel4cp` verification. Solid arrows: formally verified correspondence as part of this work, red arrows indicate correspondence proofs not yet complete at time of writing; dashed arrows: unverified/unimplemented. The dark green box represents the abstract spec agains which the seL4 kernel's implementation was verified.

5 VERIFICATION PIPELINE

The automated verification of the seL4CP implementation proceeds via multiple stages (see Figure 2) and reuses a number of tools and libraries developed for the seL4 kernel verification effort. This is not just for convenience: it reduces the risk of semantic mismatch, where the assumptions of one artifact in the proof chain may not be satisfied by the guarantees of the previous. Such mismatches can be subtle and easily obscured by unverified correspondences.

5.1 Verifying `libsel4cp`

The C implementation of `libsel4cp` is first processed by the same C parser that is used in seL4 kernel verification [Klein et al. 2014]. This tool translates the C source code into the SIMPL programming language [Schirmer 2006], which has a well-defined operational semantics in Isabelle/HOL. This allows us to use the very same C semantics as the kernel verification. In the next step, we perform a semantics-preserving translation of the SIMPL code into a control-flow graph in the SydTV-GL language, using the existing `SimplExport` tools.

SydTV-GL is a common intermediate language, able to represent essentially arbitrary, unstructured control flow. It is already used in the binary verification of the seL4 kernel [Sewell et al. 2013] (and should eventually enable extending the `libsel4cp` correctness proof to the binary). We manually transcribe the seL4CP Haskell specification (Section 4.2) into the quantifier-free theory of arrays and bit vectors (`QF_ABV`) implemented in SMTLIB2, a standard input-output format for SMT solvers. Figure 3 shows how the various specifications are connected.

We have developed a new tool, Gordian, to verify the C code represented by the control flow graph. Specifically, Gordian takes the SMTLIB2 `QF_ABV` specification, as well as the exported SydTV-GL graph, annotates the graph with the specification, then generates a single logical verification condition, using a variant of the weakest-precondition calculus for unstructured programs of Barnett and Leino [2005]. This verification condition is then passed to an SMT solver, which either proves the condition or provides a counter-model. If the verification condition is proved successfully, the implemented C function satisfies its specification.

On a successful run, the verifier establishes that the given C functions are *functionally correct with respect to the provided specification*, and provides strict guarantees about the absence of common programming errors including:

- null pointer dereferences and inappropriate memory accesses;
- arithmetic overflows and exceptions (signed integer overflows, division-by-zero, invalid bit shifts, invalid conversions);
- undefined behavior such as using the values of uninitialised local variables.

The SydTV-GL Export of `libsel4cp` consists of 3,540 lines of code. The Gordian tool is able to verify the functional correctness of the main handler loop of the seL4CP (condition $\star$ above). Using the `z3` SMT solver as the main backend, this verification takes about 20 seconds on a desktop computer. The proof is fully automated, no manual proof effort is required. In experiments, we found the tool resilient to code changes: when tested on multiple (correct and incorrect) minor variations of the handler loop, it always produces the expected result (verification or counter model).

5.2 Verified system initialisation

Any properties proved about the system will only hold if it is correctly initialised. We can achieve this by using a formally verified system initialiser.

CapDL is a language for describing access rights in seL4-based systems [Kuz et al. 2010]. 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, making CapDL an extremely powerful and versatile tool for managing seL4-based systems. There are multiple tools which can initialise seL4-based systems into a given CapDL distribution: these include the original `capdl-loader`, written in C and maintained by the Trustworthy Systems Group, and a new Rust CapDL loader [Spinale 2023]. There is also a *formally verified system initialiser*, called `case-init` for historical reasons, which is written in the CakeML language that has a verified compiler [Tan et al. 2016].

We augment the seL4CP SDK with functionality for automatically generating `CapDL` language output corresponding to the system specification. Since the SDK is written in

Python, we can reuse the well-tested and maintained preexisting Python CapDL bindings for this development.

Since it is not possible to formally verify the functional correctness of the Python SDK directly, we implement *translation validation*: a correspondence between the input SDF and the output CapDL is shown *post hoc*, in each instance. We accomplish this by transcribing a part of the implementation relation into Isabelle/HOL. The input SDF and the output CapDL are both imported into Isabelle/HOL (one as the static platform configuration, the other as the kernel state), and a simple automated proof script verifies that the implementation relation holds between them, excluding the part of the relation pertaining to the contents of virtual address spaces.

The verified CapDL already allows us to use either the `capdl-loader` or the `rust-capdl-loader` to initialise seL4 Core Platform systems.[2] The formally verified `case-init` tool does not support the 64-bit Arm architecture yet, but once ported, the seL4CP will have fully verified system initialisation.

6 THREATS TO VALIDITY

We only prove functional correctness between the induced semantics for the C code and its specification, so the compiler and linker need to be trusted. This gap may be bridged by combining our tool with the existing seL4 binary-verification toolchain, which ensures that the seL4 kernel proofs apply to the compiled kernel binaries. We do not yet verify the absence of stack overflows.

The proof work is done by SMT solvers: we therefore assume that the SMT solver used is functionally correct and is invoked correctly. Since we thoroughly test and check the correctness of the different transformations in the Gordian tool, we trust that it generates the correct SMT queries required to prove condition $\star$, but we have not formally verified it.

We have to trust that the `sel4cp` tool correctly exports its SDF input to Isabelle/HOL. In practice, this is a straight forward data transformation which is amenable to manual inspection.

As always, one also has to make some bottom-level assumptions about the physical world and other code running in the system: these have to be left to future work (where possible) or have to be validated by empirical means. If these assumptions are not met, faults can still occur. In our case, the assumptions are that the hardware works as specified, the kernel has been loaded correctly, and that the libraries outside the scope of the verification project, such as `libseL4`, also satisfy the properties stated in the spec. The correct initialisation by `sel4cpinit` is assumed unless the verified `case-init` is used.

[2]The latter requires a fork of the seL4 Core Platform that does not use the mixed-criticality scheduling features of the kernel.

7 CONCLUSIONS

Our newly developed Gordian verification tool has successfully verified the functional correctness of the main handler loop of `libseL4`. We also have achieved a verified mapping of the system specification to the CapDL formalism, which allows using the verified `case-init` system verifier (subject to resolving compatibility issues). Thanks to the use of SMT solvers, the proofs are fully automated, and no manual proof effort is required.

All artefacts discussed in this paper are open-sourced, see https://trustworthy.systems/projects/TS/sel4cp/verification/.

ACKNOWLEDGMENTS

We thank Ben Leslie for developing seL4CP, Nick Spinale for the Rust CapDL loader, and the UK National Cyber Security Centre (NCSC) for financial support. Most of all, we gratefully acknowledge the contributions of former Trustworthy Systems member Zoltan A. Kocsis, who is the brain behind the work but was unable to be listed as an author due to his personal stand against mandatory ORCID registration.

REFERENCES

Mike Barnett and K. Rustan M. Leino. 2005. Weakest-Precondition of Unstructured Programs. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. Lisbon, PT.

Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. *Commun. ACM* 9 (1966), 143–155.

Gernot Heiser, Lucy Parker, Peter Chubb, Ivan Velickovic, and Ben Leslie. 2022. Can We Put the "S" Into IoT?. In *IEEE World Forum on Internet of Things*. Yokohama, JP.

C.A.R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21 (1978), 666–77.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM Transactions on Computer Systems* 32, 1 (Feb. 2014), 2:1–2:70.

Ihor Kuz, Gerwin Klein, Corey Lewis, and Adam Christopher Walker. 2010. capDL: A Language for Describing Capability-Based Systems. In *Asia-Pacific Workshop on Systems (APSys)*. New Delhi, India, 31–35.

Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. 2018. Scheduling-Context Capabilities: A Principled, Light-Weight OS Mechanism for Managing Time. In *EuroSys Conference*. ACM, Porto, Portugal, 14.

Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. 2019. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *EuroSys Conference*. ACM, Dresden, DE, 15.

Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling symbolic evaluation for automated verification of systems code with Serval. In *ACM Symposium on Operating Systems Principles*. 225–242.

Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button

Verification of an OS Kernel. In *ACM Symposium on Operating Systems Principles*. ACM, 252–269.

Lucy Parker. 2022. The seL4 Device Driver Framework (sDDF). Talk at the seL4 Summit. https://sel4.systems/Foundation/Summit/abstracts2022#a-sDDF

Sean Peters, Adrian Danis, Kevin Elphinstone, and Gernot Heiser. 2015. For a Microkernel, a Big Lock Is Fine. In *Asia-Pacific Workshop on Systems (APSys)*. ACM, Tokyo, JP, 7.

Norbert Schirmer. 2006. *Verification of Sequential Imperative Programs in Isabelle/HOL*. Ph.D. Dissertation. Technische Universität München.

Thomas Sewell, Magnus Myreen, and Gerwin Klein. 2013. Translation Validation for a Verified OS Kernel. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Seattle, Washington, USA, 471–481.

Nick Spinale. 2023. Rust-seL4 CapDL Loader. https://gitlab.com/coliasgroup/rust-seL4/-/tree/main/crates/capdl

Yong Kiam Tan, Magnus Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2016. A New Verified Compiler Backend for CakeML. In *International Conference on Functional Programming*. Nara, Japan, 14.

Trustworthy Systems. 2023. The seL4 Core Platform. https://trustworthy.systems/projects/TS/sel4cp/