cyberagentur

ECOSYSTEM FOR TRUSTWORTHY IT LOS 3: FORMAL VERIFICATION OF HARDWARE-SOFTWARE-INTERFACES

Christoph Lüth,
Dieter Hutter,
Milan Funck,
Jan Zielasko

Version 1.0 (Final Version) | 31st January 2023

Herausgeberin:
Agentur für Innovation in der Cybersicherheit GmbH

Disclaimer

Impressum

Herausgeberin: Agentur für Innovation in der Cybersicherheit GmbH
Große Steinstraße 19, 06108 Halle (Saale), Germany
E-Mail: kontakt@cyberagentur.de
Internet: www.cyberagentur.de
Twitter: https://twitter.com/CybAgBund

Die Nutzungsrechte liegen bei der Herausgeberin.

Lizenz: CC BY-NC-ND 4.0: https://creativecommons.org/licenses/by-nc-nd/4.0/
Erscheinungsdatum: 12.07.2023
Redaktion: Abteilung Sichere Systeme, Referat Sichere Hardware und Lieferketten

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Ecosystem for Trustworthy IT (Ökosystem vertrauenswürdige IT)

Lot 3: Formal Verification of Hardware-Software-Interfaces

Christoph Lüth, Dieter Hutter, Milan Funck, Jan Zielasko

Version: 1.0 (Final Version)

31st January 2023

Abstract

We report on the state of the art in the area of formal verification at the hardware-software interface. Following a systematic survey of the literature covering the last ten years, we can give a landscape of existing verified systems and verification tools. Verification methods have come of age, with verified operating system kernels, compilers, and hardware, but there are still substantial gaps, in particular with respect to security; we detail these research needs and propose directions for future research efforts.

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
Cyber-Physical Systems
Bibliothekstraße 5
28359 Bremen
https://www.dfki.de/cps

Contents

1 Introduction

For a system to be *trustworthy*, it is a necessary precondition that the complete system is correct, from the applications running on the system down to the hardware. Correctness, as understood here, means that there is a formal specification of the intended behaviour of the system, to which the system adheres.

This comprehensive notion of correctness (the "whole stack" or end-to-end) requires interfaces and abstractions between the different layers of the system. Figure 1 shows the typical layers of abstraction in a system (inspired by [85]): a system is grounded in the physics underlying its electronics. From simple gates and circuits, a CPU is built with a specific microarchitecture. The most prominent abstraction layer is between the hardware and the software. This interface is defined by the *instruction set architecture* (ISA) of the hardware, which defines how the hardware behaves as seen by the software. In the software, we abstract more and more from the concrete implementation, first by an operating system which hosts user applications (including possibly more layers such a hypervisor between the hardware and the operating system), second by a virtual machine on which applications run. They form the interface of the system to the outside world, the latter being either the human user or an environment in which the system acts, or a network of other machines.

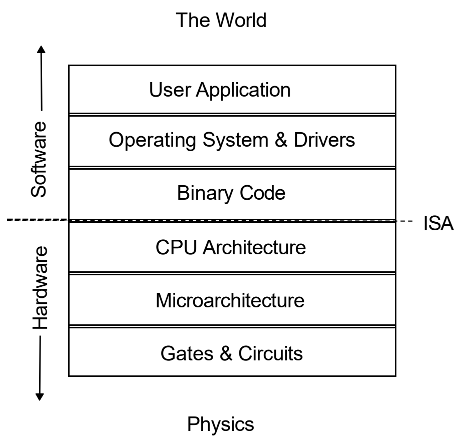


Figure 1: Layers of abstraction.

Structuring the system in such a way raises the question how correctness, which is usually stated at the user-visible upper levels (the "world"), can be transferred and proven at the lower levels. This question is exacerbated by the vastly different languages and development paradigms in hardware and software. This makes formal verification at the hardware-software interface challenging.

The *scope* of this report lies on the challenges we face when formally verifying the correctness of combinations of hardware and software. We are concerned with formal proofs, rather than exhaustive testing or hardware-in-the-loop tests, and our focus is on the layers in Figure 1 which are not greyed out; in other words, the correctness of low-level system software (operating system, device drivers, hypervisors) and the ISA, but not correctness of the microarchitecture, nor correctness of software applications, which are covered by other lots. However, the question how to cross the boundary between hardware and software and verify correctness properties of a complete system is also relevant here.

The structure of this report is as follows: Section 2 revisits the basics of formal verification, with a focus on specific challenges of verification at the hardware-software interface. Section 3 presents the results of a systematic survey of the state of the art, where we have surveyed and evaluated the publications of the last ten years in relevant conferences. Finally, Section 4 details future research needs, and proposes a roadmap for future efforts.

2 Formal Verification

Trustworthy systems have to be safe against threats from the outside (attacks) and from the inside (defects). The former is broadly subsumed as security, the latter as safety.

2.1 Safety and Correctness

Typical safety properties state the system satisfies a given specification. The specification can pertain to the required functional behaviour of the system, *i.e.* specifying which response is expected to which input, or the non-functional behaviour, which specifies response time, power consumption, memory consumption (for software) or die area (for hardware); consequently, we speak of *functional* and *non-functional correctness*. Functional correctness can be stated in terms of conditions on the input, or starting state of the system, implying conditions on the output, or end state of the system; these can be formulated in a variety of logics (see below), and take the form of contracts or assume-guarantee pairs. Going beyond these simple mechanisms, we can formulate requirements in formal modal logics such as LTL or CTL [134]. Non-functional properties are formulated either directly ("System must always respond within 20 μs."), or by extending specification mechanisms with the required dimension (*e.g.* timed automata, duration calculus, hybrid automata [5, 40, 88]). Typically, violations of such safety conditions can be eventually detected by monitoring the behaviour (*e.g.* run-time checks) of a system [4]. This will change if we consider security requirements.

2.2 Security

An important class of non-functional properties are *security properties*. Classical security properties like confidentiality or integrity are typically enforced by access control, or information flow control mechanisms. Ultimately, an access-control policy regulates for each pair of subject (process) and object (data) which actions are allowed to be executed. Access control policies are represented in a structured way, for instance, by means of roles [62] or capabilities [57] to ease their formalization and maintenance. While access control policies regulate the access to data, they do not consider the information stored in the data. In contrast, information flow control policies regulate the flow of information between different subjects (or security domains) within a program. Hence, instead of controlling the access to confidential data, information flow control regulates the effect of confidential information to the (visible) behaviour of a system. *E.g.*, whenever running a program twice with same public input (but different secret ones), the visible behaviour of both runs are indistinguishable. Data hidden to an observer must not affect the (part of the) behaviour of the system seen by the observer. As a result, the violation of such requirement can only be detected by comparing different runs of a system. In theory, they represent closure properties on sets of possible system runs (called hyperproperties) [45], which are out of range for a run-time monitor to check.

In an operating system, for instance, a kernel has to isolate different processes from each other to ensure that actions of one process can only influence other processes in an authorized way. However on a hardware level, resources, like memory or I/O-devices, are typically shared between various processes. Uncontrolled access of processes to shared resources would enable unauthorized information flow between them using the common resource as a covert channel (Information filed into the resource by one process is read by the other process). Hence, access to resources are usually implemented via a virtualization of (physical) resources hiding their real physical addresses and therefore preventing that a process can circumvent the access control. Furthermore, virtualization must hide any dynamic property of a resource that is influenced by the behaviour of other processes from the process (providing temporal/spacial separation). Typical examples are the response time when accessing memory (caching!) or the occupancy of a device by other processes. Since communication between processes requires the ability of at least one process to change the stored information in the resource, shared access to static information like physical locations of devices do not constitute a covert channel.

2.3 Structuring Mechanisms

To build large systems, we need to be able to structure both the development and the properties of the system into smaller constituent pieces, and introduce well-defined interfaces between them. For these structuring operations, we distinguish between:

- *Horizontal structuring* decomposes a complex system into smaller components, and includes modularization into components, aggregation of similar components, composition and decomposition.
- *Vertical structuring* links models on different abstraction levels by refinement (or abstraction in the inverse direction).

For verification, the interesting question is whether structuring operations preserve the property in question, *i.e.* whether we can prove properties of the whole system by proving properties of the constituent parts, and combine them. This is called *compositionality*. In general, safety properties are compositional, but security properties are not (although the specifics depend on the particular properties: functional correctness of sequential programs is compositional, but neither non-functional correctness nor correctness of concurrent programs).

The idea of *refinement* is to construct a model of the system at a higher level of abstraction, where we can show the required properties, and then derive a system model at a lower level where the properties are preserved. This technique is used quite often in verification, often implicitly. For example, the verification of the seL4 kernel is conducted with respect to an abstract model written in a functional language; this can be seen as a refinement from an abstract model in a high-level language down to a concrete model in an efficient low-level language [113].

There are two ways to work with refinements: one can either construct two models, and show *post facto* that the concrete model refines the abstract one, or one can construct the concrete model from the abstract one step-wise, by applying refinement operations which are guaranteed to preserve correctness [15]. Related to this is the idea of *cross-level verification* [82, 35], where different abstraction levels are used simultaneously to achieve greater coverage, by relating properties stated at the abstract level to the implementation provided on a lower level.

As mentioned above, refinement allows one to translate valid (safety) properties about an abstract system to concrete properties about its implementation by applying a series of correctness-preserving transformations. In contrast to the safety case, security properties specified and proven in an abstract system do not uniformly translate to valid properties about a realized system. Roughly speaking, security properties relate the abilities of an (unknown) adversary to successfully attack the system. Each refinement will provide the adversary with a new (refined) vocabulary, which they can arbitrarily use to formulate an attack, while any attack mentioned on the abstract level will be refined, *i.e.* implemented to a particular attack in the refined layer.

2.4 Challenges of the Hardware-Software-Interface

Verification on the hardware-software-interface can be challenging, because it combines two very different computational paradigms, worlds views and resulting development flows.

Hardware is costly to produce, and immutable. This means that errors are very costly to fix (for example, Intel's infamous Pentium FDIV bug resulted in a cost of 475 million dollar[1]). Subsequently, hardware development is very engineering-driven, with an emphasis on correctness at the cost of flexibility; errors are not tolerated. Software, on the other hand, is flexible, can be produced at hardly any cost once developed, and can be changed reasonably easily afterwards. This results in a more agile development methodology, which puts an emphasis on flexibility and change; errors are tolerated, and corrected once detected.

Hardware has a smaller state space compared to software (*e.g.* a fixed set of registers as opposed to a large number of variables), making it more amenable to automatic proof methods such as model checking and symbolic execution via SMT-provers. Software has a larger state space, and also a larger vertical development space— the abstraction levels range from the ISA over low-level software (device drivers, operating system) to user applications. The hardware of a system is typically described in one language (a hardware description language, HDL, such as VHDL or Verilog), whereas a number of languages are used for the software — from the binary machine code described by the ISA over low-level languages such as C to higher-level languages such as Java or Javascript (ECMAScript) to domain-specific languages describing particular application areas, reflecting the vertical design space described above.

One problem when combining different formalisms is that we have to guarantee *consistency*, in the sense that shared symbols are given the same meaning, and that no contradictions are generated. This is typically achieved by embedding the formalisms into a more general one

[1]Source: Intel Corporation 1994 Annual Report, via Wikipedia.

(specification languages like UML, first-order or higher-order logic, type theory *etc.*), or by a well-defined interface linking both sides (the prime example of this being the ISA).

Beyond that, when combining hardware and software we have to consider the properties of the system *as a whole*, which combines both worlds multiple times. For example, a typical application has a software stack (user-interfacing application code running on a Java virtual machine, running in an operating system which serves as an interface to the hardware), and properties are formulated at the abstract level (*e.g.* a credit card number entered in the application must remain private), but have to be proven across multiple abstraction layers (here, from application code down to micro-processor executing binary machine code, and the keyboard-driver actually recording the number as it is entered). Fortunately, the ISA is a well-defined interface between hardware and software, but it may need to be extended for the case at hand (*e.g.* when considering peripherals such as the keyboard). Thus, we need a methodology how to transfer properties between hardware and software — from the software world to the hardware and back. This discussion links with the compositionality from previous Section 2.3; here, linking software applications to hardware is a vertical structuring operation, and we are interested which properties are preserved.

In general, as mentioned before, safety properties (defined as properties of each run of the system) are preserved from the software to the hardware, but security properties (defined, for instance, as a closure property on the set of possible system runs) are not. Precisely because hardware is so radically different from software, a whole new Pandora box of threats opens up. New covert channels are available on the hardware level, based on execution time, power consumption, race conditions, or even caching behaviour inside CPU-cores that potentially disclose evidence of code execution to a hostile observer and thus open new avenues for side-channel attacks. Software compartments being the main aids in security to separate individual activities must be mapped in terms of temporal or spatial separability on a hardware level giving rise to the need to thoroughly monitor and control provenance and flow of information on all channels even partially observable by an adversary.

2.5 Verification Methods and Tools

Verification methods and tools need to handle these challenges. In general, the question whether a given system satisfies a given property is undecidable; it becomes decidable once we limit ourselves to systems with a finite state space. Then, verification needs to consider all possible system runs (or states) to check that the required properties are satisfied. However, the number of states (the state space) becomes too huge as to make the question practically undecidable for realistic systems. There are three principal tools which help us to handle this problem: The first is *compositionality*, which allows us to decompose the state space of the overall system into the smaller state spaces of its components; the second is *abstraction*, which allows us to reduce the state space while preserving the properties to be shown; and the third one is *automated theorem proving*, which shows a property not by state enumeration but by logic inference rules.

Unfortunately, state abstraction is very much dependent on the particular property; it can be used as abstract interpretation to great effect in *e.g.* showing that no exceptions or no integer overflows occur [139]. Model checking is a general term for a family of techniques where the system is modelled as a finite state machine, with a smaller state space than the original one, properties are formulated in variations of temporal logic (LTL, CTL, timed logics) and proven by state exploration. This technique can be efficient for safety properties, and useful because it can provide concrete counter-examples, but in general awkward to show absence of faults.

Theorem proving can be divided into automated theorem proving, where the logical inference rules are applied without user guidance, and interactive theorem proving, where the human user guides the rule application. Automated theorem proving works well with decidable fragments of certain logics, *e.g.* Pressburger arithmetic with linear inequalities as used in SMT provers (or solvers) such as Z3. These provers are very powerful in their niche, but often used invisibly as verification engines behind the scene in verification tools; there is even a standard interface language SMTlib to allow interchangeability between these tools.

Modern interactive provers such as Isabelle or Coq make use of automated provers by using them to prove trivial lemmas which are then combined into more comprehensive results. Isabelle has a sophisticated methodology for this (called Sledgehammer [27]), which uses automated provers to explore the state space, and reconstructs the proof inside Isabelle, without compromising the provers' consistency. Table 1 shows an overview, without any claim of completeness, over the existing proving tools, as used in the state of the art as surveyed in Section 3.

Symbolic execution is state exploration combined with theorem proving, by expressing the state evolution as a logical formula. Originally developed for software [110] modern tools such as KLEE [36] can be very efficient when considering moderate to large state spaces. Related to this is the technique of predicate abstraction, where predicates over integers are reduced to equivalent predicates over boolean variables [18, 44, 65].

In the context of verification in the hardware-software interface, the methodology of *virtual prototyping* [55, 129, 90] is relevant as well. A virtual prototype is an implementation of the hardware in software, in particular of the microprocessor but comprising the whole system including peripherals as well, and using a high-level systems modelling language such as SystemC or SpinalHDL. Originally developed for early software development to enable software development before hardware is physically available, virtual prototypes can be used for verification purposes as well, *e.g.* by using symbolic execution to show correct behaviour [34, 194].

Name	URL, Details
Interactive provers	
Isabelle/HOL[a]	https://isabelle.in.tum.de/ Typed higher-order logic with automated prover support and tactics, rich user interface, large library of available theories.
Coq	https://coq.inria.fr/ Inductive calculus of constructions, dependent types, rich library of available theories. Recipient of the 2013 ACM Software System Award
K	https://kframework.org/ Rewrite-based semantic framework for the definition of formal operational semantics
ACL2	https://www.cs.utexas.edu/users/moore/acl2/ Inductive first-order logic with term rewriting. Recipient of the 2005 ACM System Software Award
HOL4	https://hol-theorem-prover.org/ Typed higher-order logic. Slightly dated but still in widespread use.
Alloy	http://alloytools.org/ Relational logic (as in UML). Fully automatic, with integrated model finder.
Maude	http://maude.cs.illinois.edu Rewriting logic engine
Dafny	https://dafny.org/ Programming language with integrated specification language and program verifier
Automatic provers and model checkers	
Z3	https://github.com/Z3Prover/z3, SMT prover
Alt-Ergo	https://alt-ergo.ocamlpro.com/, SMT prover
Yices	https://yices.csl.sri.com/, SMT prover
Spin	https://spinroot.com Model checker, Recipient of the 2002 ACM System Software Award.
nuSMV	https://nusmv.fbk.eu/ Model checker for synchronous and infinite-state systems

[a]Isabelle is a generic theorem prover, and Isabelle/HOL the specific instance of Isabelle for classical typed higher-order logic; this is the only Isabelle logic ever used in this field, so here when we refer to Isabelle, we mean Isabelle/HOL.

Table 1: Overview of theorem proving tools used in verification.

2.6 Summary

- We distinguish safety and security properties. The former include functional and non-functional correctness, and can be decided in terms of input and output of a single run; the latter include confidentiality, privacy, and integrity, and are not decidable on a single system run.
- Horizontal structuring operations construct the system from smaller components, and vertical structuring operations relate models of the system at different levels of abstraction (refinement). Compositional properties are preserved by horizontal structuring; safety properties are compositional, whereas security properties are not.
- Verification at the hardware-software interface is a challenge because of the different computational paradigms and world views (languages, tools, development flows). This makes it hard to transfer properties, in particular security properties, from software to hardware or vice versa. The central interface of the system is the interface from the hardware to the software, the instruction set architecture (ISA).

3 State of the Art

To systematically survey and assess the existing literature, we have reviewed all papers in the last ten years from leading conferences (and journals) in the field of formal verification, looking for papers in the area of hardware-software verification. Specifically, we were interested in

- formalization and verification of ISAs,
- verification and development of low-level software (as defined above),
- and tool support for these activities.

The specifics and statistics about our approach can be found in Appendix A.

3.1 Formalization and Verification of ISAs

Formalization of an ISA means the construction of a model in a format which can be read and processed by a computer. We have found 12 papers concerned with this endeavour. The languages used for this purpose can be diverse; they range from programming languages or HDLs such as Haskell or System Verilog to theorem provers such as the K framework, ACL2, Isabelle, or Coq, to domain-specific languages (DSLs) invented for this specific purpose, such as SAIL and its precursors. Most industrially relevant ISAs have been formalized, including ARMv8,

Name, Refs	ISA	Language	Notes
[178]	x86	HOL4	Model of x86-TSO in HOL4
[140, 53]	x86	K	Fully executable semantics, uncovered bugs in manual
RockSalt [143]	x86	Coq	Checking software-based fault isolation ACL2 [75]
CORANA [197]	ARM	Z3	Extracts formal semantics from natural language spec. for symbolic execution
[75]	x86	ACL2	Verification of x86 machine code
[69]	ARMv7	HOL4	Formalizes ARMv7 ISA
[166]	ARMv8	DSL	Describes formal language ARM uses to specify ARMv8 architecture
GRIFT [176]	RISC-V	Haskell	Sequential simulation, coverage analysis
riscv-formal [204]	RISC-V	System Verilog	Formal testbenches for RISC-V cores
[182]	RISC-V	Kami	Used by SiFive
SAIL [9]	Generic	DSL	Not tied to specific ISA, used for ARMv8, RISC-V, MIPS

Table 2: Overview of recent ISA formalization efforts.

x86, MIPS, and RISC-V. The formalization efforts are summarized in Table 2. One point to note is that the x86 ISA proved so complex that there has been work to learn its behaviour from running programs [91, 195].

The first objective of the formalization efforts was to obtain an unambiguous and precise description of the behaviour of the hardware, which the software running on the machine can rely on. This was then used as a basis for verification (see next section), but also to check consistency of the specification [207], to check conformance of hardware designs to the ISA [204], to model the micro-architecture [67], to verify machine code running on that architecture [75, 143, 135] or compilers targeting it [106].

A prominent recent development is SAIL, a domain-specific language used to specify instruction set architectures. Its precursors can be found, *inter alia*, in the language used by ARM to specify their ARMv8 architecture [68, 166], but it has since then been used to specify the ARMv8 architecture, MIPS, and RISC-V; for the latter, it has been adopted as the ‘gold standard’ by RISC-V international. The value of SAIL is that ISA specifications are completely unambiguous, readable for engineers, and can be used to generate code for theorem provers or other analysis tools. Figure 2 gives an overview of the SAIL architecture and tool landscape. From the generic description of an ISA, the SAIL tools can generate sequential emulators. Moreover, the Isla tool [11] proves symbolic execution for SAIL ISA specs, which is used in [11] to generate tests from a specification (taking the weak memory semantics into account). The Islaris tool [174] uses the symbolic exection engine together with the Iris formalization of separation logic in Coq to verify ARMv8 and RISC-V machine code. Via translation to the Lem language [146], a prover-independent higher-order language to model semantics, defini-

Name, Refs	ISA	Language	Notes
ISA-Formal [167]	ARMv8	DSL, Verilog	Industrially used at ARM
Forte [105, 152]	x86 (i7)	DSL[a]	Industrially used at Intel
[72]	RISC-V	Cadence Jasper, BlueSpec Verilog	Verification focuses on CHERI extensions to ISA
[93]	SPARCv8	Isabelle	Proves non-interference for LEON3
[75, 54, 76]	x86	ACL2	Industrial verification of an embedded x86 processor at Centaur
Bedrock [61]	RISC-V	Kami	Simple core, part of stack covering whole embedded system
CSLED [208]	x86-32	Coq	Derives consistent instruction encoder/decoder

[a]Combines own theorem prover with model checking (symbolic trajectory evaluation).

Table 3: Overview of recent ISA verification efforts.

tions of the ISA in Coq, Isabelle and HOL4 can be obtained. (Circumstantial evidence suggests that these generated models are not easy to work with.) SAIL has furthermore been used to specify the CHERI architecture, an experimental extension of standards ISAs with so-called *capabilities* (see Section 3.7).

Verification of an ISA means ensuring that an implementation of the ISA in hardware (the micro-architecture) behaves as formally specified. As opposed to software, in hardware formal verification — meaning *proving* that the micro-architecture behaves as *formally* specified — is more established. Table 3 gives an overview about recent ISA verification efforts, as they appear in the literature; note entries by Intel [105] or ARM [167] using formal verification as part of their design flow. It is further stated [72] that "formal verification methodologies for [RISC-V instructions] are well-established", so it is safe to assume that formal verification in industrial practice is well established and not always published (an observation corroborated by personal experience), and hence more wide-spread than Table 3 suggests. Still, there are two main problems: firstly, verification which is not published may be of use to the manufacturer but not to the user — in particular, a formal model of the ISA to be verified against must be available, such that users can base software verification on it. This is not always the case, *e.g.* for the x86 ISA. Secondly, most of the papers and efforts are concerned with functional correctness of the ISA; this is sufficient for safety properties, but verification of security properties may require properties beyond what the ISA can express ([12] is a step in this direction).

A substantial body of work is concerned with languages to model hardware; just like the programming languages for software, hardware description languages (HDL) are evolving constantly. Recent HDLs in this direction include Kami [42], which is integrated into the Coq theorem prover, and allows seamless correctness proofs; Chisel [14] and SpinalHDL [153] (the latter a fork of the former) which evolved from the RISC-V community at Berkeley, and Clash [13], which are built on functional languages with all the advantages that come with them;

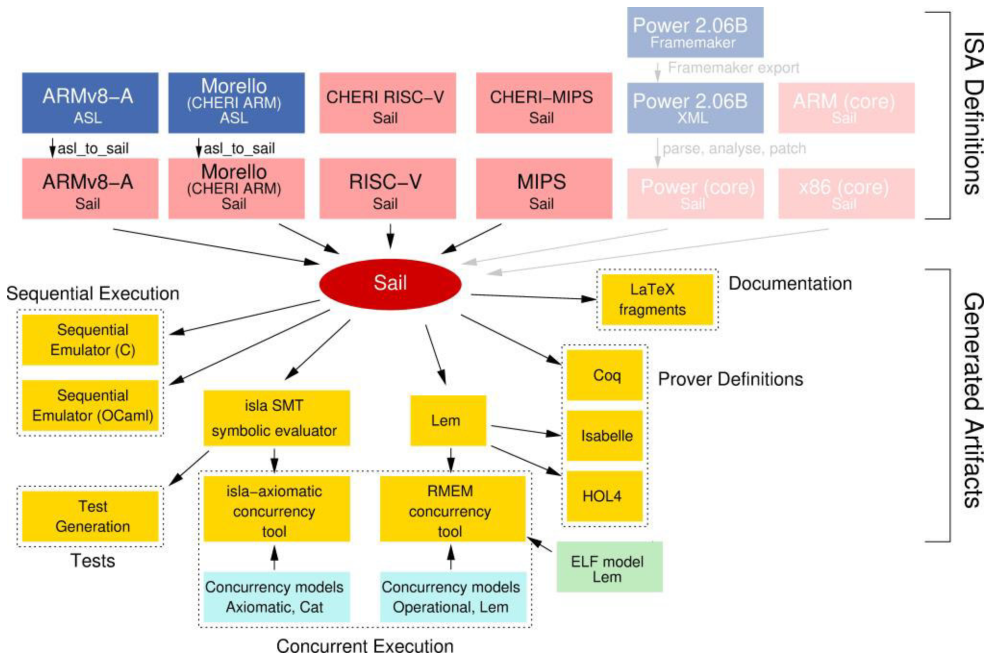


Figure 2: Overview of the SAIL architecture and tool landscape.

or more recent languages such as PDL [210] targeted at pipelined processors.

3.2 Memory Models

A surprising amount of work concerns memory models, *i.e.* the behaviour of memory as observed from the software. This is partly because memory access is typically either not part of the ISA, or only in a very simplified fashion. The typical assumption [2] is "a per-core virtual address space translated, at page granularity via a memory management unit (MMU), to a single global physical address space containing all the random access memory (RAM) and memory mapped devices in the system." However, in typical modern SoCs the situation is far more complex; Figure 3 shows the block diagram of the (at time of writing slightly dated) TI SoC OMAP 4460 (quoted in [2]); as we can see, addresses can be remapped in a multitude of ways, even creating loops if configured incorrectly. The paper [2] presents an embedding of configuration models for a memory bus into Isabelle, and allows *e.g.* consistency proofs to exclude such pathological situations; other work [190, 191] addresses the question how such complex memory models impinge on the formal verification of programs using them.

This situation is mirrored on the software side by the memory model of programming languages, in particular C. The C standard views memory as "objects, composed of contiguous sequences of one or more bytes."[50, Section 6.2.6.1 (2)], and does not even assume one contiguous address space. All modern compilers and operating systems assume a more detailed model here, making use of "implementation-defined behaviour" as permitted but not described by the standard. This means that verifications are always specific to a particular configuration of machine architecture and compiler used. A number of papers [138, 127, 142] propose

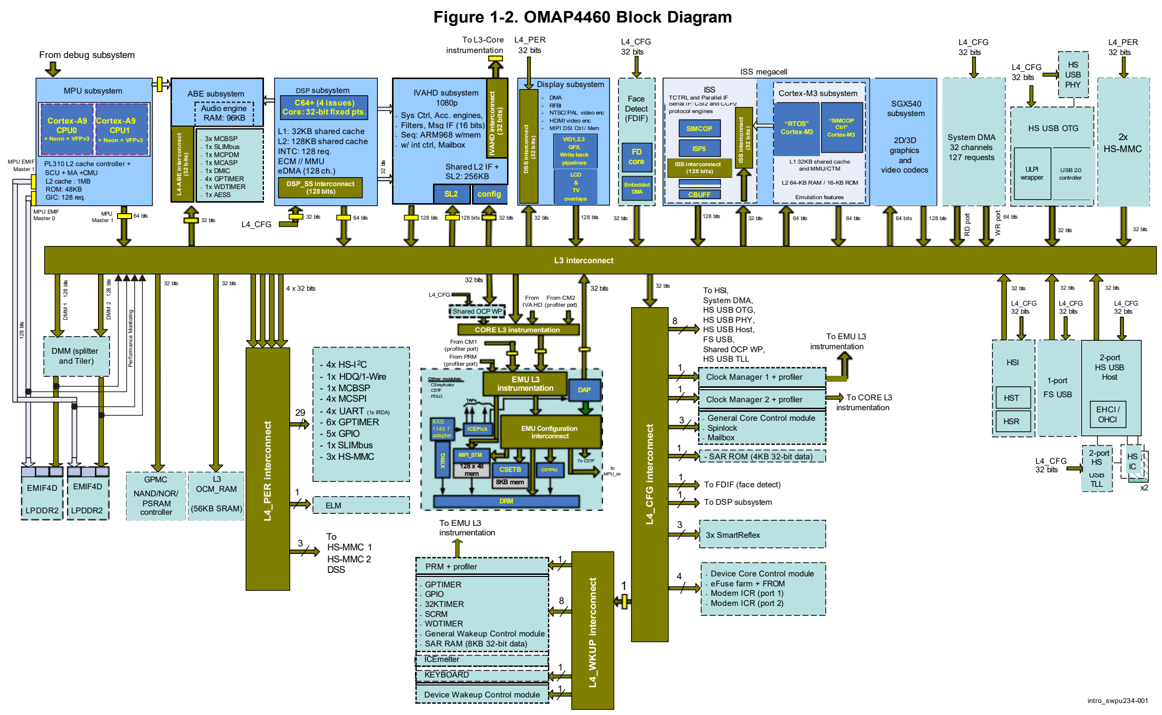


Figure 3: Block diagram of the TI OMAP 4460.
©Texas Instruments, Source https://www.ti.com/lit/ug/swpu235ab/swpu235ab.pdf

extensions which allow the portable verification of realistic programs (such as OS kernels).

Other papers address the problem of concurrency in the presence of weak memory models. Modern processors rarely access the memory in precisely the sequence as given by the program, creating additional problems (for example, the recent Spectre/Meltdown bugs were caused by speculative execution, where memory was accessed before the program reached the corresponding instructions). One way out of this problem is by total store ordering (TSO); [163, 109] formalize this for x86 architectures. [162] presents an approach allowing to explore other such models formally, and [48, 132] propose DSLs to build memory models complete with reasoning support, [60] even with a purpose-built model checker, and [96] with a focus on hardware extensions. Languages and provers used here include Isabelle, Alloy, Maude, and Dafny.

However, in summary it is fair to say that finding memory models which are both abstract enough to allow efficient verification of software utilising the memory and at the same time precise enough to capture the hardware behaviour accurately enough remain an active research area.

3.3 Programming Languages

For all its faults [117, 138], C is the dominating language when writing low-level software. C is portable, well-known, and has excellent and diverse tool support. This is reflected in the literature: in our survey, we have found 13 papers concerned explicitly with the C language (not counting those verifying software written in C), three with assembly language, two with Rust, and one each with Scala, Lustre, and Esterel. As far as the foundations go, the established consensus is the use of some variant of Hoare logic, extended with separation logic to handle references and pointers.

Apart from papers discussing deficiencies of the language, in particular the standard[2], and suggesting remedies [138, 175, 127], the main bulk of this work was concerned with verified compilation. A milestone in this area was the first verified C compiler CompCert [128] by Leroy and others, who have been awarded the 2021 ACM System Award for this achievement. Subsequent work and papers have built on this, for example:

- The Verified Software Toolchain [6, 7] by Appel and others uses CompCert to build a tool chain which guarantees correctness from the source down to the machine-language program running on a weakly-consistent-shared-memory machine, much in the spirit of Sir Tony Hoare's verifying compiler [92].
- Various papers are concerned with extensions to CompCert to include separate compilation [188, 107], linking [164], or inclusion of assembly language [186] (as is common in low-level code).
- A fork of CompCert is used as the compiler for the verified operating system kernel CertiKOS [80, 81].
- Other extensions cover "stack awareness", *i.e.* ensuring programs do not run out of stack space [198, 38].

A different approach is to define a DSL (*e.g.* RESOLVE [165, 192] or CIVL [86]) which is expressive enough to cover interesting programs without the entrenched difficulties of a long-established language, but compared to work on C this seems to have been rarely followed up.

Apart from C, the language Rust has recently gained a lot of traction [101], but there is not much work (yet) on large verification efforts using Rust, even though the RustBelt project [100, 51] has laid the necessary foundations for this. An interesting recent development is to extend C with ownership types (RefinedC, [175]), combining established tool support for C with one of the main innovations of Rust.

Furthermore, there is work on verifiable assembly languages [30, 70, 21, 151], which make use of the comparatively simple structure of assembly language to automate proofs, but this seems a rather niche application (*e.g.* Vale [30, 70] is used in the context of cryptography). Notably, we did not find many references using Ada or SPARK [19] apart from [71], even though it is an established framework to develop high-quality software, suggesting its use is in the aerospace domain rather than system software. Similarly, Esterel [66] and Lustre [32] are used in the context of real-time systems, and Scala [121] is not a systems programming language.[3]

[2]There is some overlap here with the work on memory models discussed in Section 3.2, as these are a part of the standard, and hence the language, and in fact one of its main deficiencies.

[3]For the same reason, the substantial body of work concerning verification of the Java platform, including the

3.4 Verification of System Software

Early work on the verification of operating systems dates back to the eighties (see [112, Table 2] for an overview), but these did not concern realistic kernels. Recent efforts include seL4 and CertiKOS; Table 4 also shows a number of industrial entries, demonstrating that formal software verification has arrived in the main stream. This also means, as discussed above, that there may be industrial verification projects which have not been published, *e.g.* it is unclear whether Microsoft actively maintains the proofs for the Hyper-V hypervisor, and there are rumours about verification activities at Apple. On the other hand, the start-up Prove& Run has announced a verified kernel [28, 29] but not published any details or papers about this work.

The two verified OS kernels, CertiKOS and seL4, exhibit the two main contemporary approaches to kernel development. CertiKOS[4] is the *Certified Kit Operating System*. It is not a micro-kernel, but a full kernel with scheduler, file system, and various hardware drivers. It uses an intricate layer-based framework to guarantee correctness of the whole kernel, where specifications (and components) can be layered on top of each other. Components can be written in C or assembler; a fork of the CompCert compiler covers correctness of the linking process as well, and guarantees correctness of the whole kernel. All proofs are done in the Coq theorem prover. In contrast, seL4[5] is a microkernel, derived from the L4 family of microkernels. It formally proves the isolation of applications running on the system in Isabelle/HOL. There is an ecosystems of tools surrounding seL4, *e.g.* the CAmkES and sDDF framework which allow driver development for seL4 [155], but these are not part of the formal correctness proof of seL4.

The verification of device drivers started in earnest with the use of predicate abstraction [18, 17]. Predicate abstraction is a technique to reduce the state space of a program to make them amenable to model-checking, while preserving the validity of the properties in question, such as program safety, memory leaks or conformance to an API protocol. Tools like C2BP [18] (part of the SLAM suite) or BLAST [181] implement the technique, and are used for static driver verification in Windows [16] and Linux [161] respectively, although the latter seems not actively maintained any more. There has been work on formalising network protocols (TCP, [169]) or file systems ([170]); more recent efforts include mCertiKOS [41] or Vigor [157].

One prominent tool suite for the verification of C programs is Frama-C [111] (even though it was not used in any of the papers we have surveyed). In Frama-C, specifications are added as specially formatted comments (annotations) in the E-ACSL language. It combines various static analysis techniques with the verification condition generator Why3 [63] to prove these assertions. It is actively maintained and used in various industrial projects [23].

programming language and the JVM, has been omitted from this survey, as Java is not a systems programming language. It might be interesting to consider a verified JVM as a virtualization layer, allowing user applications only on this virtual machine (as in the Android OS); there is work such as [83] (which we did not explore) in this direction.

[4]http://flint.cs.yale.edu/certikos/framework.html

[5]https://sel4.systems

Name and Refs	Prover	Notes
VeriSoft [47]	Isabelle, VCC	Used VCC to show separation property of Microsoft Hyper-V hypervisor; not actively maintained any more
seL4 [113, 179]	Isabelle	Proves functional correctness and security properties
CertiKOS c[81]	Coq	Certified abstraction layers for correctness and safety
SyberX [207]	Isabelle	CC evaluation to EAL5+
NOVA [24]	Coq	Proves confinement properties for NOVA hypervisor
OSEK [56]	nuSMV	Proves correct configuration of customised kernels
N.N. [39]	Iris, Coq	Proves correctness of IPC algorithms for new microkernel at Meta
VERVE [209]	Coq, Boogie	Verifies correctness at the binary level by compiling to typed assembly language and showing refinement from abstract model in Coq

Table 4: Kernel verification efforts.

3.5 Verification of Systems

A substantial body of work is concerned with the verification and validation of complete embedded systems; application highlights include the control software for the Chinese lunar rover [180], and an ERTMS Railway train spacing system [43]. If industrial standard languages are used for verification, they need to be given a formal semantics, *e.g.* AADL[6] [126, 125] or the UML [168]. However, most of these efforts focus on the software, constructing a model of the software which can be verified [89, 160, 64, 3]. Another approach to verified control software is to generate code, often using control theory [154, 89, 122]; this is lacking a hardware angle and so is not considered further in this report. Another left-field area is verification of CUDA kernels [145, 58]; although this is low-level according to our definition, we consider CUDA not to be essential components of trustworthy systems.

When verifying complete systems, we need to consider consistency across the hardware-software border, *i.e.* when verifying the correctness of the ISA against its specification, and verifying the correctness of the software running on that ISA, we need to be sure that both specifications of the ISA are at least consistent. Not a lot of work has been concerned with that aspect; in [61] a verification of this kind is carried out for a simple embedded system, embedding all the specifications inside one common framework (*viz* the Coq prover). It seems debatable whether this approach scales to larger systems (with an operating system, multiple drivers, and user applications running on it).

3.6 Security

The previous verification efforts are mostly concerned with functional correctness or program safety properties. Covering security we review a few papers investigating protection against

[6]Architecture Analysis & Design Language, an SAE standard for the avionics industry

Name and Refs	Prover	Notes
Microsoft SDV [16]	SLAM	Uses predicate abstraction and static analysis to verify safety of Windows device drivers
Linux SDV [161]	BLAST	SDV for the Linux eco-system
mCertiKOS [41]	Coq	CertiKOS covering device drivers and interrupts.
SybilFS [170]	Lem	Provides a model of POSIX file system, derives test orfacles
[169]	HOL4	Provides high- and low-level models of TCP and sockets API, proves correctness, provides validation tests.
PerSeVerE [118]	model-checker	Formalizes semantics of ext4 file system, can model-check programs to use its API correctly.
[157]	Vigor	Formal verification of a NAT stack.

Table 5: Driver verification efforts

side-channel attacks before detailing the CHERI approach.

The Scam-V tool [147] allows validating observational models (using symbolic execution) against real hardware, to detect side channels by generated test programs. [59] uses SMT solvers for a similar objective, showing data is perfectly masked. In contrast, [185] formally shows (using Isabelle) threat security (absence of data leaks) for weak memory models, in particular for ARM architectures. [49] develops a general methodology (in Coq) for the mCertiKOS verified operating system for security proofs, using a general notion of observation for *e.g.* state indistinguishability; this approach covers the whole software stack, but does not cover the underlying hardware. [87] describes an approach to prevent timing channel attacks, both on the hardware and software side. This has been implemented and evaluated for seL4 [73] for the x86 and ARM processors, and for RISC-V [203] leveraging additional hardware support. It is currently being verified [184].

SAFE [12] is a "clean-slate design for highly secure computer systems", comprising both hardware and software, with formal end-to-end proofs of non-interference. The hardware considered in SAFE is assumed to have capabilities beyond what one would usually expect from an ISA, much in the spirit of CHERI.

3.7 Capability Hardware Enhanced RISC Instructions (CHERI)

The CHERI approach [205, 202] extends standard ISAs with architectural capabilities that allow for a fine-grained access control of memory on a HW-level. These capabilities are unforgeable tokens of authority supporting a highly scalable software compartmentalization as well as the realization of a secure (*i.e.* monitored) software (C or C++) pointers.

In principle, capabilities relate to portions of the address space and define the permissions that owners of the capabilities possess on them. They can be marked as invalid (withdrawing the associated permissions) or as sealed (disabling any modifications and usage). Capabilities are protected by the architecture of the ISA. They can only be constructed by instructions that do so explicitly and new capabilities cannot exceed the permissions or address space of the capabilities underlying its creation. In particular, this gives rise to a security property of capability monotonicity on code execution. The set of capabilities accessible to the code cannot increase during its execution (preventing privilege escalations) unless execution is yielded to another domain.

While at boot time the firmware is equipped with initial capabilities covering the entire address space, in each stage of the software stack (boot loader, hypervisor, operating system, processes, etc.) address space and permissions of the related capabilities will become more restrictive in accordance to the monotonicity property. Another property is the intentionality of credentials, such that if a process passes a capability as an argument to a system call, the kernel can use only that credential to guarantee that no other process memory covered by other capabilities of the kernel is accessed. To deal with capabilities, CHERI introduces special and general capability registers and adds new ISA instructions operating exclusively on capabilities in these registers (ensuring intentionality!).

CHERI also supports the use of C/C++ language and virtual-memory-based software by a hybrid capability architecture to integrate a capability model with a conventional MMU-based architecture. In particular, it constrains integer-related memory access by a default data capability and instruction fetches by a program-counter capability.

CHERI extensions to 64-bit MIPS and 32/64-bit RISC-V ISA are specified in the SAIL-language (*cf.* Section 3.1) and are available as open source[7]. These SAIL pecifications are the starting points to derive reference documentations, ISA-level simulators in C or OCaml, hardware instruction tests as well as emulators and prover definitions of the architecture for a formal verification in Isabelle/HOL, COQ or HOL4 (*cf.* Figure 4).

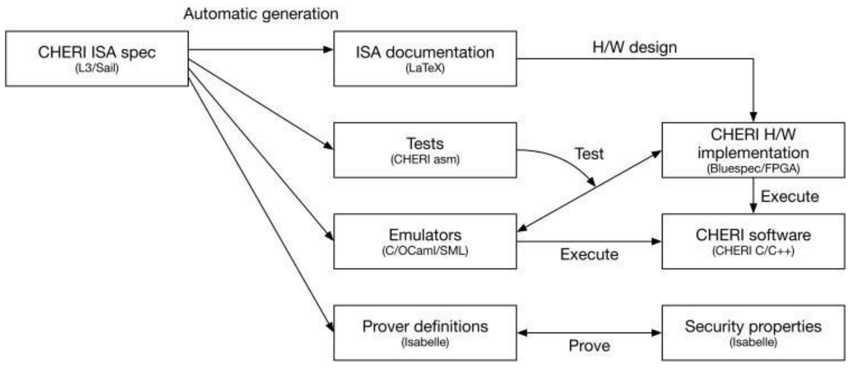


Figure 4: Main artefacts of the CHERI engineering process as in [202].

The CHERI extensions have been formally verified using the Isabelle prover with respect

[7]CHERI-MIPS: https://github.com/CTSRD-CHERI/sail-cheri-mips,
CHERI-RISCV: https://github.com/CTSRD-CHERI/sail-cheri-riscv

to the various security properties [149]. These are in particular, 1. that they satisfy the monotonicity and intentionality properties, 2. that arbitrary code cannot change system registers or memory without having explicit permissions to do so, and 3. guarantees about the execution of untrusted code in a controlled isolation boundary.

There are two main use cases of CHERI capabilities with respect to C/C++ based software stacks concerning fine-grained memory protection and scalable software compartmentalization.

Fine-grained memory protection. The use of capabilities instead of integers to implement C/C++-language pointers, and minor extensions to operating system and language runtime-system enable a strong and efficient spatial, referential, and temporal memory safety for these traditionally memory-unsafe languages. CHERI allows for two compilation modes. In a pure-capability mode, in particular all pointer types (of C/C++) are realized by capabilities. Then, the architecture preserves the integrity and the provenance of the pointers, thus preventing an injection of pointer values from outside or an in-memory corruption of pointers. Narrowing the bounds of the address space and restricting permission (typically done by the compiler) prevent pointers from being abused for purposes other than for what they were intended.

Scalable software compartmentalization. Following the MILS-paradigm [173], conventional MMU-based software compartmentalization decomposes larger software applications into components running in isolation and communicating only in a controlled way. Capabilities open up an alternative means to construct the software isolation and controlled communication required to implement compartmentalized software designs. Unlike MMU-based compartmentalization (*i.e.*, implemented using virtual memory), capability-based techniques allow for more granular and scalable data sharing, as they are restricted by page-granularity sharing and the utilization of multiple address spaces and thus the number of compartments.

CHERI comes with an adapted reference stack for its architecture realizing the benefits of capabilities. This includes extended versions of the real-time operating system FreeRTOS and the BSD-based operating system FreeBSD as well as its own CheriOS microkernel demonstrating in particular granular compartmentalization together with strong memory safety. Common developer tools like Clang/LLVM compiler, LLD linker and GDB debugger are extended to support the CHERI architecture. Finally, various extended user-space libraries and applications are also available.

3.8 Summary

The state of the art can be summarized as follows:

- Established instruction set architectures are the x86 and ARM families, and more recently RISC-V. For all of these, formalizations and verification support are available. The SAIL language and framework allow convenient tool development for all ISAs.
- C is the *lingua franca* of system development, and has been investigated thoroughly. There is a verified C compiler, CompCert, and a number of verification tools such as Frama-C, or specialized theories for Isabelle or Coq.
- There are at least two verified operating systems, seL4 and CertiKOS, which have left the prototype stage and are actively supported.

The state of the art covers all layers of interest in Figure 1. We consider all tools to be at least at TRL 5 as defined in the Horizon 2020 work programme[8]. However, there are still open questions which need to be adressed. This will be the subject of the final section.

4 Research Needs and Roadmap

4.1 Research Needs

In Figure 5, we have mapped the results of our survey in the last section to the system layers in Figure 1. Note that the major points for a trustworthy system in the hardware-software region are covered — there are verified operating system kernels, a verified compiler for the most widely used language, C, and three major instruction set architectures which have been formalized and verified (marked in green in Figure 5). However, we can also see that there are necessary parts of a system which are not verified (these are marked in red). The most relevant gaps here include:

- *hardware drivers*: CertiKOS integrates drivers into the kernel and has support for the verification of the driver and the whole kernel; it remains to show this scales well in practice. seL4 handles drivers outside the trusted kernel, relying on the kernel for correctness.
- *memory buses*: there is a lot of work on memory buses, but it needs to be extended to cover complex memory protocols and configurations as found in complex SoCs and systems.
- *peripheral devices*: there is scant work on handling connection to peripheral devices, *e.g.* hard disks, but we cannot simply disregard these.

[8]http://ec.europa.eu/research/participants/data/ref/h2020/wp/2014_2015/annexes/h2020-wp1415-annex-g-trl_en.pdf

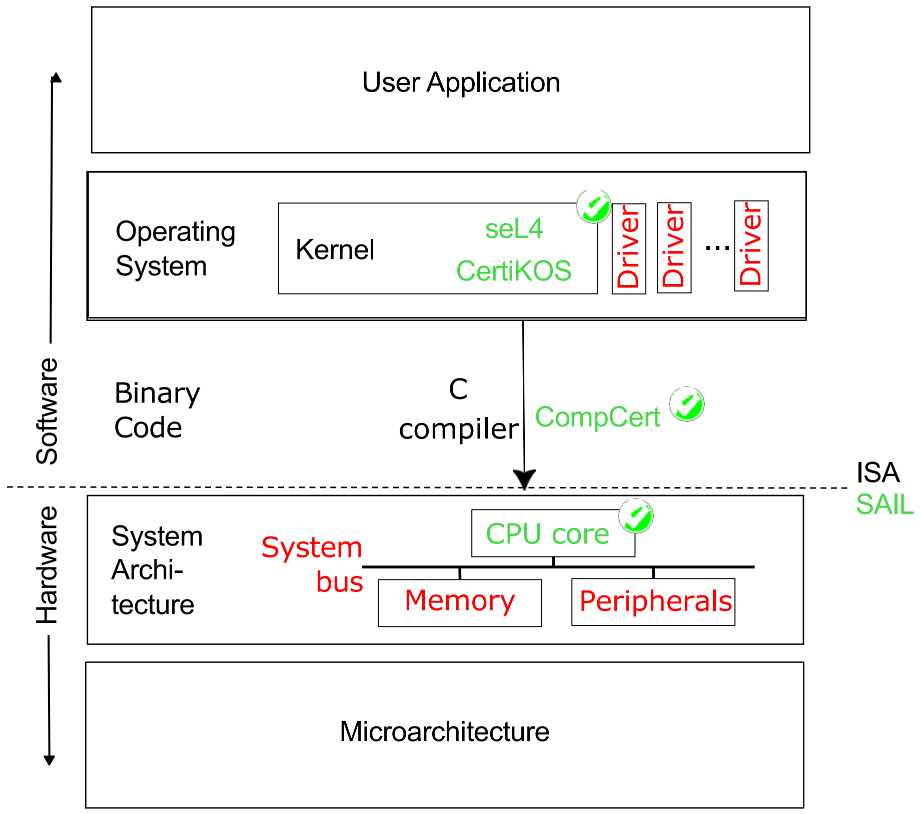


Figure 5: Trustworth systems and tooling landscape.

Moreover, the verified parts are as yet not coupled together in a comprehensive manner, nor is there a systematic way how the verification can connect to verified parts in the hardware (microarchitecture), or user applications. For the microarchitecture, while we know how to specify the ISA and verify its functional correctness, the connection of the SAIL language to hardware description languages, in particular innovative new ones like Chisel or SpinalHDL, needs improvement. The connection to user applications needs more work: essentially, it needs a comprehensive specification of what an operating system does. Of course, seL4 and CertiKOS have formal specifications, but it is unclear (and unlikely) if they are sufficient. The situation gets even more interesting when we consider security aspects, such as confidentiality, privacy or compartmentalization. Proving such properties formally will require more comprehensive specifications, and specification formalisms covering the whole system stack, possibly even across the software-hardware border; the CHERI extension built on SAIL (*cf.* Section 3.7) is a good starting point in that direction.

A further question to consider is *licensing*. Most of the tools and components mentioned above are available under open-source licenses. We posit that open-source licenses are crucial for a trustworthy environment, as they allow users and developers to analyse and scrutinize the source code to assess its correctness. In theory, it might be sufficient to provide a formal specification of the interface along with a formalized proof of correctness, but this has not been shown to work with substantial developments or security properties. On the hardware side,

Verified OS kernel	seL4, CertiKOS
Verified C compiler	CompCert
ISA specification and verification	SAIL
Verified CPU cores	diverse RISC-V cores
Provers	Isabelle, Coq

Table 6: Suggested best-of-breed tools and system components.

proprietary IP such as ARM cores might be of more use, given that they are delivered in source form, and thus at least susceptible to automatic analysis techniques.

Finally, one needs to consider *long-term viability* of both the components and the development tools. The x86 ISA is by now dated and very complex, as a result of maintaining backwards compatability for fourty years; its long-term prospects are unclear.[9] The long-term prospects of ARM and RISC-V look much better (even if the latter is still lacking widespread industry acceptance). As for languages, the C language has been with us for fifty years, and will probably stay with us for another fifty. It is supported by several excellent compilers, including CompCert. Other languages such as Rust which might take up the mantle as premier system programming language are still young and one-compiler languages. This is a problem in the long run; for long-term viability we need diversity on the supply side, being able to choose among different compilers, or in the case of hardware, OEMs.[10] This lack of diversity may be a problem for the long-term viability of verification proofs as well. Both Coq and Isabelle are built using a specific compiler (Coq uses OCaml, Isabelle uses Poly/ML), with which they interact quite closely; and both provers, although open-source, are maintained by a small group of academic volunteers. None of this discussion means tools and languages such as Rust, Isabelle or Coq should not be considered, but one should be aware of these issues, and investigate possible alternatives. This in particular means to provide well-defined interfaces between components, specified in a language which is not dependent on any one tool. (SAIL is a good example of this; a framework like Frama-C and its specification language E-ACSL could be the basis for a similar rôle for software verification.)

To sum up, for the hardware-software interface we have a number of languages and tools at our disposal; Table 6 gives an overview of the best-of-breed which can serve as a basis to build an ecosystem of trustworthy systems.

4.2 Roadmap

Based on the discussion in the previous section, we will lay out a roadmap of research projects focussed on the hardware-software interface. We classify projects according to their *risk* and

[9]In particular given Intel's recent announcement of a billion-dollar investment in RISC-V https://www.electronicdesign.com/technologies/embedded-revolution/article/21216435/electronic-design-intel-launches-1-billion-fund-to-build-foundry-ecosystem-backs-risc.

[10]This makes ARM viable in the long term, because there are a lot of hardware suppliers for ARM-based hardware.

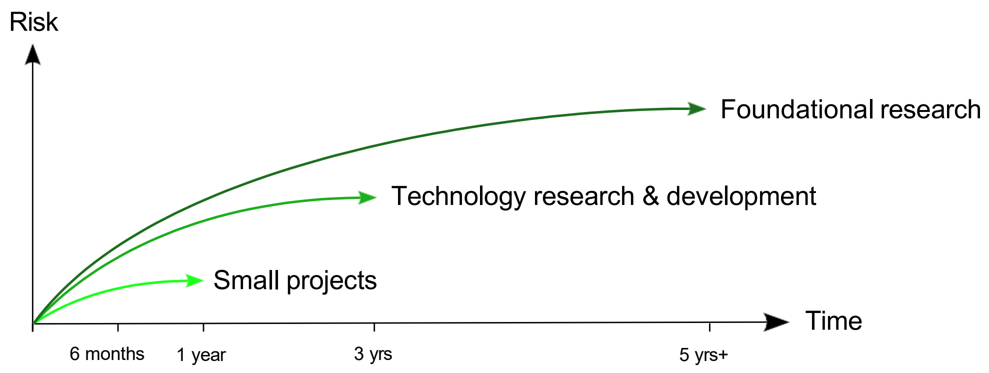


Figure 6: Classification of research projects.

the *time* horizon in three categories (Figure 6):

(i) *Small projects* have a clear technological focus, which fill gaps in the tool and component landscape. They do not require major new research or large developments. These are low-risk (it is highly likely they will succeed in the stated goal), and require something like 6 to 12 person months.

(ii) *Technology research & development* are medium-sized projects where new technologies, algorithms or tools need to be developed. They are of medium risk (they may fail, but with appropriate risk management some of the research efforts should always be recovered), and take times between 36 and 72 person months (typically, three years with one to three persons). These are the typical BMBF-funded projects.

(iii) *Foundational research* is research where at the start not much more than basic research questions are known, and the project develops foundations, methodologies and tools to solve the question. Comparable to DFG-funded projects, these are of high risk (they are as likely to fail as to succeed, but they should fail productively, providing a negative answer to the research question) and last anytime between 24 and 50 person months.

4.2.1 Small Projects

These are small projects with a well-defined scope, using existing technologies and methods. Following are a few suggestions:

(S-1) *Extend SAIL with support for functional HDLs* like SpinalHDL, Chisel or Clash. Presently, SAIL can generate tests as System Verilog Assertions, but if one wants to verify conformance of a SpinalHDL design, one needs to manually translate the SAIL specification to SpinalHDL.

(S-2) *Connect SAIL to riscv-formal and Symbiyosys.* This would integrate SAIL, and the 'gold standard' RISC-V specification in SAIL, to the existing RISC-V tool.

(S-3) *Provably correct RISC-V ISA specifications.* The RISC-V specification generated from the SAIL specification in Isabelle via Lem is not very readable or friendly to work with. In order to be able to better work with it, we can define a more succinct spec directly in Isabelle and formally show equivalence with the generated specification.

(S-4) *Rebase the seL4 verification to SAIL.* The current seL4 verification is based on the ISA modelled in Isabelle/HOL. It needs to be investigated whether it is easier to replace this ISA model with the specification of the RISC-V ISA specification from the previous project, or whether one proves equivalence of this specification and the official RISC-V ISA separately.

(S-5) *Connect SAIL to existing virtual prototypes.* There exist a number of virtual prototypes, in particular for RISC-V systems, but most of them are based on independent formalizations of the RISC-V standard; connecting them with the existing SAIL standard would give increased confidence.

(S-6) *Virtual prototyping support for CHERI.* In a similar vein, adding support for the CHERI extensions to existing virtual prototypes for RISC-V would open the way to develop software making use of these extensions before RISC-V hardware supporting CHERI is available.

(S-7) *Connect CompCert with SAIL*. The CompCert compiler has its own model of the binary code produced at the end; connecting this to existing SAIL standards would close a gap and contribute to an idea of overall correctness.

(S-8) *Extend SAIL with support for Rust*. Currently, SAIL supports C and OCaml, but support for Rust in the same form as with C would help with developing system software in Rust.

There may be more, similar projects here that will come up during the more long-term projects sketched in the following. It makes sense here to set up a framework where these projects can be applied for and granted in an agile, light-weight way, as for example with Google's Summer of Code projects. These would also serve as good student projects. As students are an import avenue of dissemination, we recommend setting aside funding for this specific purpose.

4.2.2 Technology Research & Development

Other gaps in the tool landscape require a more comprehensive effort. These are the main gaps that we have identified:

(R-1) *Comprehensive verification of a realistic embedded system*, *e.g.* a router. This would show how the tools work together, and identify further gaps in the landscape; it would also indicate how to structure a fully verified system, and how to connect the tools. As opposed to [61], this system would preferably not be implemented in one single theorem prover, but connect the different provers and formalisms from the tool landscape.

(R-2) *Safe and secure peripherals.* Components such as network stacks or file systems are essential to any operating system. There has been early work in this direction [169, 170]. A fully verified implementation of a network stack and POSIX-compliant file system is a substantial effort, and still lacking.

(R-3) *Drivers and peripherals for seL4.* There is already work on how to implement drivers for seL4, and how to extend seL4 to a full operating system ([155], *cf.* discussion above on page 15), but this has to the best of our knowledge this has not been subject to formal verification (as opposed to CertiKOS [41]). After setting up a methodology to add drivers, integration of the results of the previous project would seriously enhance seL4's verified capabilities.

(R-4) *Extend Frama-C with ownership types and capabilities.* There is research in extending C with ownership types, bringing the benefits of Rust into the C world. In this project, it should be investigated how Frama-C, and in particular its specification language can be extended to that effect; the ownership types would be the software equivalent of the hardware capabilities on the CHERI side. The project would require

- defining a suitable annotation syntax and semantics;
- adding support to Frama-C to check these annotations;
- or adding support to the prover tools (the Jessie plug-in, which is currently not actively maintained).

When successful, this would extend Frama-C as a development and verification interface for CHERI-based code.

(R-5) *A formally verified virtual prototype.* Virtual prototyping has proven a valuable tool in early software development, *i.e.* developing software for hardware which is under development. However, none of these have been formally verified to the extent that *e.g.* CompCert has. Such a formally verified VP would be a valuable addition to the ecosystem.

(R-6) *A formally verified Rust microkernel.* In order to diversify the ecosystem in terms of language support, develop a fully verified microkernel in the Rust language. This is a good test for the tool support of Rust, in particular the verification support (the Rustbelt tool) and if the fragment of Rust covered by verification support is large enough. Of particular interest would be how to integrate external modules into the kernel, as Rust can distinguish between safe and unsafe code; it would be favourable if driver code could always be safe.

4.2.3 Foundational Research

Beyond filling gaps, there are some issues that require a more foundational approach.

(X-1) *Comprehensive security.* Much of the discussion and projects in the previous section, as much of the previous work on formal verification from Section 3, is concerned with functional, and to a lesser extent, non-functional correctness. The software tool stack (as well as its underlying basis in hardware) is aligned to efficiently support the realization of a given functionality but provides only few means (if any at all) to explicitly constrain an (automated) refinement process by demanding non-functional or security requirements. An example would be to demand that information stored in particular variables must not influence the generation of communication messages. In this situation, a compiler should adjust the intended memory allocation of variables within a program according to the formulated security needs and in line with the security measures provided by the hardware. CHERI is a good start in this direction, and can serve as an anchor to develop a tool stack which provides means for formally verified security. The main task would be to lift the guarantees provided by the hardware via the ISA into security notions on higher level (user applications). This includes the development of a vocabulary of security notions on the programming level and of compiling techniques to refine these notions into sufficient requirements on an ISA-level. Subsequent steps include investigating if CHERI is powerful enough to support these aspects, or needs extending. There is also the aspect of extending the verification tools in the landscape to handle CHERI. The end vision here is a system with formally verified secure enclaves.

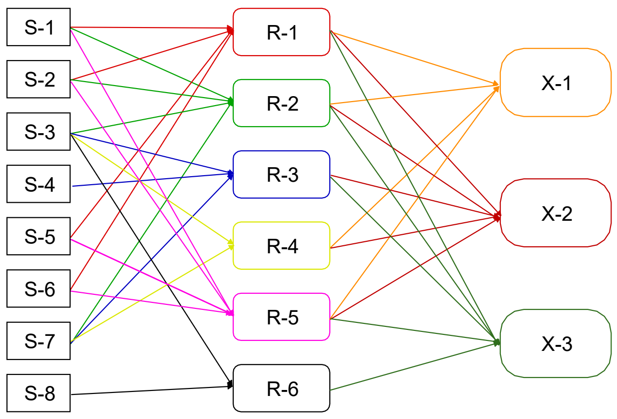
(X-2) *Reference architecture for trustworthy systems.* In the long run, the actual components of a safe and secure, fully verified computing landscape are not as important as the interfaces between them. Single components will always become out of date over time, and may need to be replaced, but the basic interfaces will remain stable. This concerns

- the hardware-software interface (the ISA),
- the memory bus and other busses connecting peripheral devices to the CPU,
- the interface of the kernel to the various driver modules,
- the interfaces of major operating systems (such as file systems and network stacks), and
- the interface of the operating system kernel to the user applications.

It extends beyond the components to the tools. A reference architecture should specify specification and modelling languages, independent of the tools (provers) used.[11]

The reference model should include comprehensive tests, and where possible automatic prover tools, for all components, in particular for drivers, as in the Microsoft SDV. This is particularly relevant, as drivers are a frequent source of erroneous behaviour (in particular, system crashes).

[11]One will probably need to fix the programming language for practical reasons, but should strive for multi-language support.



Project	depends on
R-1	S-1, S-2, S-5, S-6
R-2	S-1, S-2, S-3, S-7
R-3	S-3, S-4, S-7
R-4	S-3, S-7
R-5	S-1, S-2, S-5, S-6
R-6	S-3, S-8

Project	depends on
X-1	R-1, R-2, R-4, R-5
X-2	R-1, R-2, R-3, R-4, R-5
X-3	R-1, R-2, R-3, R-5, R-6

Table 7: Dependencies between proposed research projects.

(X-3) *Proof engineering:* how to structure and conduct huge formal proofs. A formally verified ecosystem of trustworthy systems relies on formal proofs, but there is not much work as yet on the construction of such proofs. Early pioneering efforts such as the proofs of the Four Colour theorem, the Kepler conjecture or the Feit-Thompson theorem have been heroic efforts by sole individuals rather than concerted group efforts, as opposed to the situation in hardware or software development, where *e.g.* the Linux operating system has seen countless contributors over the years.

There have been a number of papers and efforts to remedy this situation [114, 171, 148, 99], but most of these are specific to particular provers or application areas. What is lacking is a general approach which covers more foundational questions, such as what makes a good prover and prover language, how can we refactor and reuse proofs, or how should one approach big formal proof efforts, aside from tools supporting these questions.

4.2.4 Dependencies and Priorities

There are dependencies among the research projects proposed in the following sections, which are summarized in Table 7. These dependencies should be seen as supportive rather than *conditiones sine quibus non*.

We also suggest the following priorities, depending on the general overarching goal:

- To *create a formally verified and secure operating system ready for the end-user as soon as possible*, we pick one candidate — most likely seL4 — and complete the missing pieces. This could then serve as the blue-print for the reference architecture.

  Priorities: R-1, R-2, R-3, R-5; X-2

- To create *an ecosystem of companies, start-ups, research institutes and university groups focussed on trustworthy IT*, a diversity and breadth of approaches need to be considered instead of focussing on one particular operating system with corresponding prover and verification methodology. Interoperability and standardized interfaces will be crucial to allow exchange of ideas, implementations, and tools.

  Priorities: R-1, R-2, R-4, R-5, R-6: X-1, X-2.

- To *establish the foundations of trustworthy IT*, a diverse approach with a more long-term perspective is needed. Verification efforts for drivers and peripherals are seen case studies by which we can study the underlying mechanisms, and how to improve our formalisms and tools.

  Priorities: R-4, R-5, R-6: X-1, X-3

4.3 Summary

- Formal software and hardware verification has left the laboraty, and can be used productively on real system software such as operating system kernels.
- However, the landscape of existing verified systems has still considerable gaps which need to be filled: hardware drivers, memory buses, peripherals, to name a few.
- In particular as far as security is concerned, verification across the hardware-software interface is still a challenge.

We propose a diverse structure of many different, perhaps even competing projects to fill these gaps instead of one central development. We posit that creating the capabilities to produce verified software, and stable interfaces between the components of an ecosystem of verified tools, creates more long-term impact than a one-off creation of one verified system.

References

[1] Reto Achermann, David Cock, Roni Haecki, Nora Hossle, Lukas Humbel, Timothy Roscoe, and Daniel Schwyn. Generating correct initial page tables from formal hardware descriptions. In *Proceedings of the 11th Workshop on Programming Languages and*

Operating Systems, PLOS ’21, pages 69–75, New York, NY, USA, October 2021. Association for Computing Machinery.

[2] Reto Achermann, Lukas Humbel, David Cock, and Timothy Roscoe. Physical Addressing on Real Hardware in Isabelle/HOL. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving*, Lecture Notes in Computer Science, pages 1–19, Cham, 2018. Springer International Publishing.

[3] Arvind Adimoolam, Thao Dang, Alexandre Donzé, James Kapinski, and Xiaoqing Jin. Classification and Coverage-Based Falsification for Embedded Control Systems. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 483–503, Cham, 2017. Springer International Publishing.

[4] Fred B. Alpern, Bowen; Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117—-126, 1987.

[5] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[6] Andrew W. Appel. Verified Software Toolchain. In Gilles Barthe, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 1–17, Berlin, Heidelberg, 2011. Springer.

[7] Andrew W. Appel. Verified Software Toolchain. In Alwyn E. Goodloe and Suzette Person, editors, *NASA Formal Methods*, Lecture Notes in Computer Science, pages 2–2, Berlin, Heidelberg, 2012. Springer.

[8] Alessandro Armando, Roberto Carbone, and Luca Compagna. SATMC: A SAT-Based Model Checker for Security-Critical Systems. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 31–45, Berlin, Heidelberg, 2014. Springer.

[9] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. The state of sail. In Matthew Fernandez, editor, *SpISA 2019: Workshop on Instruction Set Architecture Specification*, 2019.

[10] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proceedings of the ACM on Programming Languages*, 3(POPL):71:1–71:31, January 2019.

[11] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. Isla: Integrating Full-Scale ISA Semantics and Axiomatic Concurrency Models. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 303–316, Cham, 2021. Springer International Publishing.

[12] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hri¸tcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. A verified information-flow architecture. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 165–178, New York, NY, USA, January 2014. Association for Computing Machinery.

[13] C.P.R. Baaij, Matthijs Kooijman, Jan Kuper, W.A. Boeijink, and Marc o Egbertus Theodorus Gerards. CLaSH: Structural descriptions of synchronous hardware using haskell. In *Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architect ures, Methods and Tools*, pages 714–721. IEEE Computer Society, 9 2010. eemcs-eprint-18376.

[14] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1216–1225, New York, NY, USA, June 2012. Association for Computing Machinery.

[15] Back, Ralph-Johan and Wright, Joakim. *Refinement Calculus*. Texts in Computer Science. Springer Verlag, 1998.

[16] Thomas Ball, Byron Cook, Satyaki Das, and Sriram K. Rajamani. Refining Approximations in Software Predicate Abstraction. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 388–403, Berlin, Heidelberg, 2004. Springer.

[17] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Integrated Formal Methods*, Lecture Notes in Computer Science, pages 1–20, Berlin, Heidelberg, 2004. Springer.

[18] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 203–213, New York, NY, USA, May 2001. Association for Computing Machinery.

[19] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., USA, 2003.

[20] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, Lecture Notes in Computer Science, pages 364–387, Berlin, Heidelberg, 2006. Springer.

[21] Björn Bartels and Nils Jähnig. Mechanized, Compositional Verification of Low-Level Code. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods*, Lecture Notes in Computer Science, pages 98–112, Cham, 2014. Springer International Publishing.

[22] Mark Batty, Mike Dodds, and Alexey Gotsman. Library abstraction for C/C++ concurrency. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 235–248, New York, NY, USA, January 2013. Association for Computing Machinery.

[23] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The dogged pursuit of bug-free C programs: The Frama-C software analysis platform. *Communications of the ACM*, 64(8):56–68, July 2021.

[24] Hanno Becker, Juan Manuel Crespo, Jacek Galowicz, Ulrich Hensel, Yoichi Hirai, César Kunz, Keiko Nakata, Jorge Luis Sacchini, Hendrik Tews, and Thomas Tuerk. Combining Mechanized Proofs and Model-Based Testing in the Formal Analysis of a Hypervisor. In John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016: Formal Methods*, Lecture Notes in Computer Science, pages 69–84, Cham, 2016. Springer International Publishing.

[25] Jesse Bingham and Joe Leslie-Hurd. Verifying Relative Error Bounds Using Symbolic Simulation. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 277–292, Cham, 2014. Springer International Publishing.

[26] Steve Bishop, Matthew Fairbairn, Hannes Mehnert, Michael Norrish, Tom Ridge, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with Logic: Rigorous Test-Oracle Specification and Validation for TCP/IP and the Sockets API. *Journal of the ACM*, 66(1):1:1–1:77, December 2018.

[27] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT Solvers. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, Lecture Notes in Computer Science, pages 116–130, Berlin, Heidelberg, 2011. Springer.

[28] Dominique Bolignano. Applying Formal Methods in the Large. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, Lecture Notes in Computer Science, pages 1–1, Berlin, Heidelberg, 2013. Springer.

[29] Dominique Bolignano. Formally Proven and Certified Off-The-Shelf Software Components: The Critical Links for Securing the Internet of Things. Technical report, Prove & Run S.A.S., 2016.

[30] Bond, Barry, Hawblitzel, Chris, Kapritsos, Manos, Leino, K. Rustan M., and Lorch, Jacob R. Vale: Verifying High-Performance Cryptographic Assembly Code. In *26th Usenix Security Symposium*, August 2017.

[31] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and Enforcing Robustness against TSO. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 533–553, Berlin, Heidelberg, 2013. Springer.

[32] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. A formally verified compiler for Lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 586–601, New York, NY, USA, June 2017. Association for Computing Machinery.

[33] Thomas Braibant and Adam Chlipala. Formal Verification of Hardware Synthesis. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 213–228, Berlin, Heidelberg, 2013. Springer.

[34] Niklas Bruns, Vladimir Herdt, and Rolf Drechsler. Processor verification using symbolic execution: A risc-v case-study. In *Design, Automation and Test in Europe Conference (DATE-2023), April 17-19, Antwerp, Belgium*, 2023.

[35] Niklas Bruns, Vladimir Herdt, Eyck Jentzsch, and Rolf Drechsler. Cross-Level Processor Verification via Endless Randomized Instruction Stream Generation with Coverage-guided Aging. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1123–1126, March 2022.

[36] Cristian Cadar and Martin Nowack. KLEE symbolic execution engine in 2019. *International Journal on Software Tools for Technology Transfer*, 23(6):867–870, December 2021.

[37] Brian Campbell and Ian Stark. Extracting behaviour from an executable instruction set model. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*, pages 33–40, October 2016.

[38] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. End-to-end verification of stack-space bounds for C programs. In *Proceedings of the 35th ACM*

SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, pages 270–281, New York, NY, USA, June 2014. Association for Computing Machinery.

[39] Quentin Carbonneaux, Noam Zilberstein, Christoph Klee, Peter W. O'Hearn, and Francesco Zappa Nardelli. Applying formal verification to microkernel IPC at meta. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, pages 116–129, New York, NY, USA, January 2022. Association for Computing Machinery.

[40] Zhou Chaochen, C. A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, December 1991.

[41] Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward compositional verification of interruptible OS kernels and device drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 431–447, New York, NY, USA, June 2016. Association for Computing Machinery.

[42] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: A platform for high-level parametric hardware specification and its modular verification. *Proceedings of the ACM on Programming Languages*, 1(ICFP):24:1–24:30, August 2017.

[43] Alessandro Cimatti, Raffaele Corvino, Armando Lazzaro, Iman Narasamdya, Tiziana Rizzo, Marco Roveri, Angela Sanseviero, and Andrei Tchaltsev. Formal Verification and Validation of ERTMS Industrial Railway Train Spacing System. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 378–393, Berlin, Heidelberg, 2012. Springer.

[44] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-Based Predicate Abstraction for ANSI-C. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 570–574, Berlin, Heidelberg, 2005. Springer.

[45] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[46] Guillaume Cluzel, Kyriakos Georgiou, Yannick Moy, and Clément Zeller. Layered Formal Verification of a TCP Stack. In *2021 IEEE Secure Development Conference (SecDev)*, pages 86–93, October 2021.

[47] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, pages 23–42. Springer, Berlin, Heidelberg, August 2009.

[48] Robert J. Colvin and Graeme Smith. A Wide-Spectrum Language for Verification of Programs on Weak Memory Models. In Klaus Havelund, Jan Peleska, Bill Roscoe, and Erik de Vink, editors, *Formal Methods*, Lecture Notes in Computer Science, pages 240–257, Cham, 2018. Springer International Publishing.

[49] David Costanzo, Zhong Shao, and Ronghui Gu. End-to-end verification of information-flow security for C and assembly programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 648–664, New York, NY, USA, June 2016. Association for Computing Machinery.

[50] Programming langauges — C. ISO/IEC Standard 9899:1999(E), 1999. Second Edition.

[51] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. Rust-Belt meets relaxed memory. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–29, January 2020.

[52] Sandeep Dasgupta, Sushant Dinesh, Deepan Venkatesh, Vikram S. Adve, and Christopher W. Fletcher. Scalable validation of binary lifters. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 655–671, New York, NY, USA, June 2020. Association for Computing Machinery.

[53] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Ros¸u. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 1133–1148, New York, NY, USA, June 2019. Association for Computing Machinery.

[54] Jared Davis, Anna Slobodova, and Sol Swords. Microcode Verification – Another Piece of the Microprocessor Verification Puzzle. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, Lecture Notes in Computer Science, pages 1–16, Cham, 2014. Springer International Publishing.

[55] Tom De Schutter. *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press, March 2014.

[56] Hans-Peter Deifel, Merlin Göttlinger, Stefan Milius, Lutz Schröder, Christian Dietrich, and Daniel Lohmann. Automatic verification of application-tailored OSEK kernels. In *2017 Formal Methods in Computer Aided Design (FMCAD)*, pages 196–203, October 2017.

[57] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.

[58] Ariel Eizenberg, Yuanfeng Peng, Toma Pigli, William Mansky, and Joseph Devietti. BARRACUDA: Binary-level analysis of runtime RAces in CUDA programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 126–140, New York, NY, USA, June 2017. Association for Computing Machinery.

[59] Hassan Eldib, Chao Wang, and Patrick Schaumont. SMT-Based Verification of Software Countermeasures against Side-Channel Attacks. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 62–77, Berlin, Heidelberg, 2014. Springer.

[60] Michael Emmi, Rupak Majumdar, and Roman Manevich. Parameterized verification of transactional memories. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’10, pages 134–145, New York, NY, USA, June 2010. Association for Computing Machinery.

[61] Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. Integration verification across software and hardware for a simple embedded system. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 604–619, New York, NY, USA, June 2021. Association for Computing Machinery.

[62] David Ferraiolo and Richard Kuhn. Role-based access controls. In *15th National Computer Security Conference (NCSC)*, 1992.

[63] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — Where Programs Meet Provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 125–128, Berlin, Heidelberg, 2013. Springer.

[64] Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. Temporal Stream Logic: Synthesis Beyond the Bools. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 609–629, Cham, 2019. Springer International Publishing.

[65] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’02, pages 191–202, New York, NY, USA, January 2002. Association for Computing Machinery.

[66] Spencer P. Florence, Shu-Hung You, Jesse A. Tov, and Robert Bruce Findler. A calculus for Esterel: If can, can. if no can, no can. *Proceedings of the ACM on Programming Languages*, 3(POPL):61:1–61:29, January 2019.

[67] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 608–621, New York, NY, USA, January 2016. Association for Computing Machinery.

[68] Anthony Fox. Directions in ISA Specification. In Lennart Beringer and Amy Felty, editors, *Interactive Theorem Proving*, Lecture Notes in Computer Science, pages 338–344, Berlin, Heidelberg, 2012. Springer.

[69] Anthony Fox and Magnus O. Myreen. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving*, Lecture Notes in Computer Science, pages 243–258, Berlin, Heidelberg, 2010. Springer.

[70] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. A verified, efficient embedding of a verifiable assembly language. *Proceedings of the ACM on Programming Languages*, 3(POPL):63:1–63:30, January 2019.

[71] Clément Fumex, Claire Dross, Jens Gerlach, and Claude Marché. Specification and Proof of High-Level Functional Properties of Bit-Level Programs. In Sanjai Rayadurgam and Oksana Tkachuk, editors, *NASA Formal Methods*, Lecture Notes in Computer Science, pages 291–306, Cham, 2016. Springer International Publishing.

[72] Dapeng Gao and Tom Melham. End-to-End Formal Verification of a RISC-V Processor Extended with Capability Pointers. In *Proceedings of the 21st Conference on Formal Methods in Computer-Aided Design – FMCAD 2021*, pages 24–33. TU Wien Academic Press., October 2021.

[73] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time Protection: The Missing OS Abstraction. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 1–17, New York, NY, USA, March 2019. Association for Computing Machinery.

[74] Shilpi Goel. *Formal Verification of Application and System Programs Based on a Validated x86 ISA Model*. PhD thesis, The University of Texas at Austin, 2016.

[75] Shilpi Goel, Warren A. Hunt, Matt Kaufmann, and Soumava Ghosh. Simulation and formal verification of x86 machine-code programs that make system calls. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*, pages 91–98, October 2014.

[76] Shilpi Goel, Anna Slobodova, Rob Sumners, and Sol Swords. Verifying x86 instruction implementations. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 47–60, New York, NY, USA, January 2020. Association for Computing Machinery.

[77] Shilpi Goel and Jr. Warren A. Hunt. Automated code proofs on a formal model of the x86. In *Verified Software: Theories, Tools, Experiments (VSTTE'13)*, Springer, LNCS 8164, pages 222—-241, 2014.

[78] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. Don't sweat the small stuff: Formal verification of C code without the pain. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 429–439, New York, NY, USA, June 2014. Association for Computing Machinery.

[79] Jason Gross, Andres Erbsen, Jade Philipoom, Miraya Poddar-Agrawal, and Adam Chlipala. Accelerating Verified-Compiler Development with a Verified Rewriting Engine. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving (ITP 2022)*, volume 237 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:18, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[80] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 595–608, New York, NY, USA, January 2015. Association for Computing Machinery.

[81] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 653–669, USA, November 2016. USENIX Association.

[82] V. Guarnieri, M. Petricca, A. Sassone, S. Vinco, N. Bombieri, F. Fummi, E. Macii, and M. Poncino. A cross-level verification methodology for digital IPs augmented with embedded timing monitors. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 1–6, Leuven, BEL, March 2014. European Design and Automation Association.

[83] Hendra Gunadi and Alwen Tiu. Efficient Runtime Monitoring with Metric Temporal Logic: A Case Study in the Android Operating System. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods*, Lecture Notes in Computer Science, pages 296–311, Cham, 2014. Springer International Publishing.

[84] David Hardin. Instruction set architecture specification, verification, and validation using algorithmic C and ACL2. In Matthew Fernandez, editor, *SpISA 2019: Workshop on Instruction Set Architecture Specification*, 2019.

[85] Sarah L. Harris and David Harris. *Digital Design and Computer Architecture (RISC-V Edition)*. Morgan Kauffmann, 2022.

[86] Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. Automated and Modular Refinement Reasoning for Concurrent Programs. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 449–465, Cham, 2015. Springer International Publishing.

[87] Gernot Heiser, Toby Murray, and Gerwin Klein. Towards Provable Timing-Channel Prevention. *ACM SIGOPS Operating Systems Review*, 54(1):1–7, August 2020.

[88] T.A. Henzinger. The theory of hybrid automata. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, pages 278–292, July 1996.

[89] Paula Herber, Timm Liebrenz, and Julius Adelt. Combining Forces: How to Formally Verify Informally Defined Embedded Systems. In Marieke Huisman, Corina Păsăreanu, and Naijun Zhan, editors, *Formal Methods*, Lecture Notes in Computer Science, pages 3–22, Cham, 2021. Springer International Publishing.

[90] Vladimir Herdt, Daniel Große, and Rolf Drechsler. *Enhanced Virtual Prototyping: Featuring RISC-V Case Studies*. Springer, 2020.

[91] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. Stratified synthesis: Automatically learning the x86-64 instruction set. *ACM SIGPLAN Notices*, 51(6):237–250, June 2016.

[92] Tony Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. In László Böszörményi and Peter Schojer, editors, *Modular Programming Languages*, Lecture Notes in Computer Science, pages 25–35, Berlin, Heidelberg, 2003. Springer.

[93] Zhe Hou, David Sanan, Alwen Tiu, Yang Liu, and Koh Chuen Hoa. An Executable Formalisation of the SPARCv8 Instruction Set Architecture: A Case Study for the LEON3 Processor. In John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016: Formal Methods*, Lecture Notes in Computer Science, pages 388–405, Cham, 2016. Springer International Publishing.

[94] Warren A. Hunt, J. G. Carbonell, J. Siekmann, G. Goos, and J. Hartmanis, editors. *FM8501: A Verified Microprocessor*, volume 795 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994.

[95] Amjad Ibrahim, Severin Kacianka, Alexander Pretschner, Charles Hartsell, and Gabor Karsai. Practical Causal Models for Cyber-Physical Systems. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods*, Lecture Notes in Computer Science, pages 211–227, Cham, 2019. Springer International Publishing.

[96] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 703–718, New York, NY, USA, June 2022. Association for Computing Machinery.

[97] Eleftherios Ioannidis, Frans Kaashoek, and Nickolai Zeldovich. Extracting and Optimizing Formally Verified Code for Systems Programming. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods*, Lecture Notes in Computer Science, pages 228–236, Cham, 2019. Springer International Publishing.

[98] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, Lecture Notes in Computer Science, pages 41–55, Berlin, Heidelberg, 2011. Springer.

[99] Kush Jain, Karl Palmskog, Ahmet Celik, Emilio Jesús Gallego Arias, and Milos Gligoric. mCoq: Mutation analysis for Coq verification projects. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, ICSE ’20, pages 89–92, New York, NY, USA, October 2020. Association for Computing Machinery.

[100] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):66:1–66:34, December 2017.

[101] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Safe systems programming in Rust. *Communications of the ACM*, 64(4):144–152, March 2021.

[102] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’15, pages 637–650, New York, NY, USA, January 2015. Association for Computing Machinery.

[103] R. Kaivola and N. Narasimhan. Formal verification of the Pentium 4 floating-point multiplier. In *Automation and Test in Europe Conference and Exhibition Proceedings 2002 Design*, pages 20–27, March 2002.

[104] Roope Kaivola and Mark D. Aagaard. Divider Circuit Verification with Model Checking and Theorem Proving. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, pages 338–355, Berlin, Heidelberg, 2000. Springer.

[105] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, and Armaghan Naik. Replacing Testing with Formal Verification in Intel Core i7 Processor Execution Engine Validation. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 414–429, Berlin, Heidelberg, 2009. Springer.

[106] Hrutvik Kanabar, Anthony C. J. Fox, and Magnus O. Myreen. Taming an Authoritative Armv8 ISA Specification: L3 Validation and CakeML Compiler Verification. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving (ITP 2022)*, volume 237 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:22, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[107] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. Lightweight verification of separate compilation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’16, pages 178–190, New York, NY, USA, January 2016. Association for Computing Machinery.

[108] Rody Kersten, Bernard van Gastel, Manu Drijvers, Sjaak Smetsers, and Marko van Eekelen. Using Model-Checking to Reveal a Vulnerability of Tamper-Evident Pairing. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods*, Lecture Notes in Computer Science, pages 63–77, Berlin, Heidelberg, 2013. Springer.

[109] Artem Khyzha and Ori Lahav. Taming x86-TSO persistency. *Proceedings of the ACM on Programming Languages*, 5(POPL):47:1–47:29, January 2021.

[110] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[111] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, May 2015.

[112] Gerwin Klein. Operating system verification—An overview. *Sadhana*, 34(1):27–69, February 2009.

[113] Gerwin Klein. A Formally Verified OS Kernel. Now What? In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving*, Lecture Notes in Computer Science, pages 1–7, Berlin, Heidelberg, 2010. Springer.

[114] Gerwin Klein. Proof Engineering Considered Essential. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods*, Lecture Notes in Computer Science, pages 16–21, Cham, 2014. Springer International Publishing.

[115] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, June 2010.

[116] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014.

[117] Andrew Koenig. *C Traps and Pitfalls*. Addison-Wesley, 1989.

[118] Michalis Kokologiannakis, Ilya Kaysin, Azalea Raad, and Viktor Vafeiadis. PerSeVerE: Persistency semantics for verification under ext4. *Proceedings of the ACM on Programming Languages*, 5(POPL):43:1–43:29, January 2021.

[119] Abderahman Kriouile and Wendelin Serwe. Using a Formal Model to Improve Verification of a Cache-Coherent System-on-Chip. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 708–722, Berlin, Heidelberg, 2015. Springer.

[120] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. *ACM SIGPLAN Notices*, 49(1):179–191, January 2014.

[121] Viktor Kuncak. Developing Verified Software Using Leon. In Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, Lecture Notes in Computer Science, pages 12–15, Cham, 2015. Springer International Publishing.

[122] Ratan Lal, Aaron McKinnis, Dustin Hauptman, Shawn Keshmiri, and Pavithra Prabhakar. Formally Verified Switching Logic for Recoverability of Aircraft Controller. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 566–579, Cham, 2021. Springer International Publishing.

[123] Peter Lammich. Generating Verified LLVM from Isabelle/HOL. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[124] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. Coverage guided, property based testing. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):181:1–181:29, October 2019.

[125] Brian R. Larson, Patrice Chalin, and John Hatcliff. BLESS: Formal Specification and Verification of Behaviors for Embedded Systems with Software. In Guillaume Brat, Neha

Rungta, and Arnaud Venet, editors, *NASA Formal Methods*, Lecture Notes in Computer Science, pages 276–290, Berlin, Heidelberg, 2013. Springer.

[126] Jaehun Lee, Sharon Kim, Kyungmin Bae, and Peter Csaba Ölveczky. Hybrid SynchAADL: Modeling and Formal Analysis of Virtually Synchronous CPSs in AADL. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 491–504, Cham, 2021. Springer International Publishing.

[127] Rodolphe Lepigre, Michael Sammler, Kayvan Memarian, Robbert Krebbers, Derek Dreyer, and Peter Sewell. VIP: Verifying real-world C idioms with integer-pointer casts. *Proceedings of the ACM on Programming Languages*, 6(POPL):20:1–20:32, January 2022.

[128] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[129] R. Leupers, F. Schirrmeister, G. Martin, R. Kogel, T. an d Plyaskin, A. Herkersdorf, and M. Vaupel. Virtual platforms: Breaking new grounds. In *Design, Automation and Test in Europe (DATE 2012)*, pages 685–690, 2012.

[130] Tao Liu and Ralf Huuck. Case Study: Static Security Analysis of the Android Goldfish Kernel. In Nikolaj Bjørner and Frank de Boer, editors, *FM 2015: Formal Methods*, Lecture Notes in Computer Science, pages 589–592, Cham, 2015. Springer International Publishing.

[131] Francesco Logozzo, Shuvendu K. Lahiri, Manuel Fähndrich, and Sam Blackshear. Verification modulo versions: Towards usable verification. *ACM SIGPLAN Notices*, 49(6):294–304, June 2014.

[132] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada: Low-effort verification of high-performance concurrent programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 197–210, New York, NY, USA, June 2020. Association for Computing Machinery.

[133] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An Axiomatic Memory Model for POWER Multiprocessors. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 495–512, Berlin, Heidelberg, 2012. Springer.

[134] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer Verlag, 1992.

[135] Stefan Maus, Michał Moskal, and Wolfram Schulte. Vx86: X86 Assembler Simulated in C Powered by Automated Theorem Proving. In José Meseguer and Grigore Ros¸u, editors, *Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science, pages 284–298, Berlin, Heidelberg, 2008. Springer.

[136] John McLean. Security models. *Encyclopedia of Software Engineering*, 2:1136–1145, 1994.

[137] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. Exploring C semantics and pointer provenance. *Proceedings of the ACM on Programming Languages*, 3(POPL):67:1–67:32, January 2019.

[138] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. Into the depths of C: Elaborating the de facto standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 1–15, New York, NY, USA, June 2016. Association for Computing Machinery.

[139] Antoine Miné, Laurent Mauborgne, Xavier Rival, Jerome Feret, Patrick Cousot, Daniel Kästner, Stephan Wilhelm, and Christian Ferdinand. Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, Toulouse, France, January 2016.

[140] Andrew H. Miranti, Sandeep Dasgupta, and Grigore Ros¸u. Towards formalizing the x86-64 instruction decoder in K. In Matthew Fernandez, editor, *SpISA 2019: Workshop on Instruction Set Architecture Specification*, 2019.

[141] MISRA-C: 2004. Guidelines for the use of the C language in critical systems., 2004.

[142] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 187–196, New York, NY, USA, June 2013. Association for Computing Machinery.

[143] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. RockSalt: Better, faster, stronger SFI for the x86. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 395–404, New York, NY, USA, June 2012. Association for Computing Machinery.

[144] Mariano M. Moscato, Carlos G. Lopez Pombo, César A. Muñoz, and Marco A. Feliú. Boosting the Reuse of Formal Specifications. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving*, Lecture Notes in Computer Science, pages 477–494, Cham, 2018. Springer International Publishing.

[145] Stefan K. Muller and Jan Hoffmann. Modeling and analyzing evaluation cost of CUDA kernels. *Proceedings of the ACM on Programming Languages*, 5(POPL):25:1–25:31, January 2021.

[146] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: Reusable engineering of real-world semantics. *ACM SIGPLAN Notices*, 49(9):175–188, August 2014.

[147] Hamed Nemati, Pablo Buiras, Andreas Lindner, Roberto Guanciale, and Swen Jacobs. Validation of Abstract Side-Channel Models for Computer Architectures. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 225–248, Cham, 2020. Springer International Publishing.

[148] Pengyu Nie, Karl Palmskog, Junyi Jessy Li, and Milos Gligoric. Deep Generation of Coq Lemma Names Using Elaborated Terms, April 2020.

[149] Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1003–1020, May 2020.

[150] Rishiyur S Nikhil. Forvis: A Formal RISC-V ISA Specification. https://github.com/rsnikhil/Forvis_RISCV-ISA-Spec, 2019.

[151] Matt Noonan, Alexey Loginov, and David Cok. Polymorphic type inference for machine code. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 27–41, New York, NY, USA, June 2016. Association for Computing Machinery.

[152] John O'Leary, Roope Kaivola, and Tom Melham. Relational STE and theorem proving for formal verification of industrial circuit designs. In *2013 Formal Methods in Computer-Aided Design*, pages 97–104, October 2013.

[153] Charles Papon. Spinal hardware description language. https://spinalhdl.github.io/SpinalDoc-RTD/master/index.html.

[154] Junkil Park, Miroslav Pajic, Insup Lee, and Oleg Sokolsky. Scalable Verification of Linear Controller Software. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 662–679, Berlin, Heidelberg, 2016. Springer.

[155] Lucy Parker. The seL4 Device Driver Framework (sDDF). In *seL4 Summit 2022*, Munich, October 2022.

[156] David A. Patterson and John L. Hennesy. *Computer Organization and Design (RISC-V Edition)*. Morgan Kaufmann, 2021.

[157] Solal Pirelli, Arseniy Zaostrovnykh, and George Candea. A Formally Verified NAT Stack. In *Proceedings of the 2018 Afternoon Workshop on Kernel Bypassing Networks*, KBNets'18, pages 8–14, New York, NY, USA, August 2018. Association for Computing Machinery.

[158] Clément Pit-Claudel, Jade Philipoom, Dustin Jamner, Andres Erbsen, and Adam Chlipala. Relational compilation for performance-critical applications: Extensible proof-producing translation of functional models into low-level code. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 918–933, New York, NY, USA, June 2022. Association for Computing Machinery.

[159] Johannes Åman Pohjola, Alejandro Gómez-Londoño, James Shaker, and Michael Norrish. Kalas: A Verified, End-To-End Compiler for a Choreographic Language. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving (ITP 2022)*, volume 237 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:18, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[160] Elizabeth Polgreen, Kevin Cheang, Pranav Gaddamadugu, Adwait Godbole, Kevin Laeufer, Shaokai Lin, Yatin A. Manerkar, Federico Mora, and Sanjit A. Seshia. UCLID5: Multi-modal Formal Modeling, Verification, and Synthesis. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 538–551, Cham, 2022. Springer International Publishing.

[161] Hendrik Post and Wolfgang Küchlin. Integrated Static Analysis for Linux Device Driver Verification. In Jim Davies and Jeremy Gibbons, editors, *Integrated Formal Methods*, Lecture Notes in Computer Science, pages 518–537, Berlin, Heidelberg, 2007. Springer.

[162] Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur. Promising-ARM/RISC-V: A simpler and faster operational concurrency model. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 1–15, New York, NY, USA, June 2019. Association for Computing Machinery.

[163] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. Persistency semantics of the Intel-x86 architecture. *Proceedings of the ACM on Programming Languages*, 4(POPL):11:1–11:31, December 2019.

[164] Tahina Ramananandro, Zhong Shao, Shu-Chun Weng, Jérémie Koenig, and Yuchen Fu. A Compositional Semantics for Verified Separate Compilation and Linking. In *Proceed-*

ings of the 2015 Conference on Certified Programs and Proofs, CPP '15, pages 3–14, New York, NY, USA, January 2015. Association for Computing Machinery.

[165] Kalyan C. Regula, Hampton Smith, Heather Harton Keown, Jason O. Hallstrom, Nigamanth Sridhar, and Murali Sitaraman. A Case Study in Verification of Embedded Network Software. In Alwyn E. Goodloe and Suzette Person, editors, *NASA Formal Methods*, Lecture Notes in Computer Science, pages 433–448, Berlin, Heidelberg, 2012. Springer.

[166] Alastair Reid. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*, pages 161–168, October 2016.

[167] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-End Verification of Processors with ISA-Formal. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 42–58, Cham, 2016. Springer International Publishing.

[168] Daniela Remenska, Jeff Templon, Tim A. C. Willemse, Philip Homburg, Kees Verstoep, Adria Casajus, and Henri Bal. From UML to Process Algebra and Back: An Automated Approach to Model-Checking Software Design Artifacts of Concurrent Systems. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods*, Lecture Notes in Computer Science, pages 244–260, Berlin, Heidelberg, 2013. Springer.

[169] Tom Ridge, Michael Norrish, and Peter Sewell. A Rigorous Approach to Networking: TCP, from Implementation to Protocol to Service. In Jorge Cuellar, Tom Maibaum, and Kaisa Sere, editors, *FM 2008: Formal Methods*, Lecture Notes in Computer Science, pages 294–309, Berlin, Heidelberg, 2008. Springer.

[170] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. SibylFS: Formal specification and oracle-based testing for POSIX and real-world file systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 38–53, New York, NY, USA, October 2015. Association for Computing Machinery.

[171] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. QED at Large: A Survey of Engineering of Formally Verified Software. *Foundations and Trends in Programming Languages*, 5(2-3):102–281, 2019.

[172] Ian Roessle, Freek Verbeek, and Binoy Ravindran. Formally verified big step semantics out of x86-64 binaries. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, pages 181–195, New York, NY, USA, January 2019. Association for Computing Machinery.

[173] John Rushby. The design and verification of secure systems. In *8th ACM Symposium on Operating System Principles, ACM Operating Systems Review, Vol. 15, No. 5*, 1981.

[174] Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. Islaris: Verification of machine code against authoritative ISA semantics. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 825–840, New York, NY, USA, June 2022. Association for Computing Machinery.

[175] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. RefinedC: Automating the foundational verification of C code with refined ownership types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 158–174, New York, NY, USA, June 2021. Association for Computing Machinery.

[176] Benjamin Selfridge. GRIFT: A richly-typed, deeply-embedded risc-v semantics written in haskell. In Matthew Fernandez, editor, *SpISA 2019: Workshop on Instruction Set Architecture Specification*, 2019.

[177] Peter Sewell. Underpinning the foundations: Sail-based semantics, testing, and reasoning for production and CHERI-enabled architectures (invited talk). In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021, page 4, New York, NY, USA, January 2021. Association for Computing Machinery.

[178] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010.

[179] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 471–482, New York, NY, USA, June 2013. Association for Computing Machinery.

[180] Lijun Shan, Yuying Wang, Ning Fu, Xingshe Zhou, Lei Zhao, Lijng Wan, Lei Qiao, and Jianxin Chen. Formal Verification of Lunar Rover Control Software Using UPPAAL. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods*, Lecture Notes in Computer Science, pages 718–732, Cham, 2014. Springer International Publishing.

[181] Pavel Shved, Mikhail Mandrykin, and Vadim Mutilin. Predicate Analysis with BLAST 2.7. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 525–527, Berlin, Heidelberg, 2012. Springer.

[182] SiFive, Inc. Formal Specification of RISC-V ISA in Kami. SiFive, March 2020.

[183] Abhishek Singh, Rekha Pai, Deepak D'Souza, and Meenakshi D'Souza. Static Analysis for Detecting High-Level Races in RTOS Kernels. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods – The Next 30 Years*, Lecture Notes in Computer Science, pages 337–353, Cham, 2019. Springer International Publishing.

[184] Robert Sison, Scott Buckley, Toby Murray, Gerwin Klein, and Gernot Heiser. Formalising the prevention of microarchitectural timing channels by operating systems. In *International Symposium on Formal Methods (FM)*, Lübeck, Germany, 2023.

[185] Graeme Smith, Nicholas Coughlin, and Toby Murray. Value-Dependent Information-Flow Security on Weak Memory Models. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods – The Next 30 Years*, Lecture Notes in Computer Science, pages 539–555, Cham, 2019. Springer International Publishing.

[186] Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. CompCertM: CompCert with C-assembly linking and lightweight modular verification. *Proceedings of the ACM on Programming Languages*, 4(POPL):23:1–23:31, December 2019.

[187] Andrei Stefanescu. MatchC: A Matching Logic Reachability Verifier Using the K Framework. *Electronic Notes in Theoretical Computer Science*, 304:183–198, June 2014.

[188] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional CompCert. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 275–287, New York, NY, USA, January 2015. Association for Computing Machinery.

[189] Nikhil Swamy, Cătălin Hri¸tcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 256–270, New York, NY, USA, January 2016. Association for Computing Machinery.

[190] Hira Taqdees Syeda and Gerwin Klein. Program Verification in the Presence of Cached Address Translation. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving*, Lecture Notes in Computer Science, pages 542–559, Cham, 2018. Springer International Publishing.

[191] Hira Taqdees Syeda and Gerwin Klein. Formal Reasoning Under Cached Address Translation. *Journal of Automated Reasoning*, 64(5):911–945, June 2020.

[192] Aditi Tagore and Bruce W. Weide. Automatically Detecting Inconsistencies in Program Specifications. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA*

Formal Methods, Lecture Notes in Computer Science, pages 261–275, Berlin, Heidelberg, 2013. Springer.

[193] Tino Teige, Andreas Eggers, Karsten Scheibler, Matthias Stasch, Udo Brockmeyer, Hans J. Holberg, and Tom Bienmüller. Two Decades of Formal Methods in Industrial Products at BTC Embedded Systems. In Marieke Huisman, Corina Păsăreanu, and Naijun Zhan, editors, *Formal Methods*, Lecture Notes in Computer Science, pages 725–729, Cham, 2021. Springer International Publishing.

[194] Sören Tempel, Vladimir Herdt, and Rolf Drechsler. Symex-vp: An open source virtual prototype for os-agnostic concolic testing of iot firmware. *Journal of Systems Architecture: Embedded Software Design (JSA)*, 125, 2022.

[195] Freek Verbeek, Abhijith Bharadwaj, Joshua Bockenek, Ian Roessle, and Binoy Ravindran. Symbolic execution of x86 assembly in Isabelle/HOL. In Matthew Fernandez, editor, *SpISA 2019: Workshop on Instruction Set Architecture Specification*, 2019.

[196] Freek Verbeek, Joshua Bockenek, Zhoulai Fu, and Binoy Ravindran. Formally verified lifting of C-compiled x86-64 binaries. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 934–949, New York, NY, USA, June 2022. Association for Computing Machinery.

[197] Anh V. Vu and Mizuhito Ogawa. Formal Semantics Extraction from Natural Language Specifications for ARM. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods – The Next 30 Years*, Lecture Notes in Computer Science, pages 465–483, Cham, 2019. Springer International Publishing.

[198] Yuting Wang, Pierre Wilke, and Zhong Shao. An abstract stack based approach to verified compositional compilation to machine code. *Proceedings of the ACM on Programming Languages*, 3(POPL):62:1–62:30, January 2019.

[199] Jr. Warren A. Hunt and Matt Kaufmann. Towards a formal model of the x86 ISA. Technical report, Department of Computer Science, University of Texas at Austin, 2012. TR-12-07.

[200] Andrew Waterman and Krste Asanović, editors. *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*. RISC-V International, 20191213 edition, 2019.

[201] Andrew Waterman and Krste Asanović, editors. *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*. RISC-V International, 20211203 edition, 2021.

[202] Robert N.W. Watson, Simon W. Moore, Peter Sewell, and Peter G. Neumann. An introduction to CHERI. Technical Report 941, University of Cambridge, September 2019. UCAM-CL-TR-941.

[203] Nils Wistoff, Moritz Schneider, Frank K. Gürkaynak, Luca Benini, and Gernot Heiser. Microarchitectural Timing Channels and their Prevention on an Open-Source 64-bit RISC-V Core. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 627–632, February 2021.

[204] Claire Wolff. RISC-V formal verification framework. https://github.com/SymbioticEDA/riscv-formal.

[205] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. *ACM SIGARCH Computer Architecture News*, 42(3):457–468, June 2014.

[206] Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. A Practical Verification Framework for Preemptive OS Kernels. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 59–79, Cham, 2016. Springer International Publishing.

[207] Wenjing Xu, Yongwang Zhao, Chengtao Cao, Jean Raphael Ngnie Sighom, Lei Wang, Zhe Jiang, and Shihong Zou. Apply Formal Methods in Certifying the SyberX High-Assurance Kernel. In Marieke Huisman, Corina Păsăreanu, and Naijun Zhan, editors, *Formal Methods*, Lecture Notes in Computer Science, pages 788–798, Cham, 2021. Springer International Publishing.

[208] Xiangzhe Xu, Jinhua Wu, Yuting Wang, Zhenguo Yin, and Pengfei Li. Automatic Generation and Validation of Instruction Encoders and Decoders. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 728–751, Cham, 2021. Springer International Publishing.

[209] Jean Yang and Chris Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’10, pages 99–110, New York, NY, USA, June 2010. Association for Computing Machinery.

[210] Drew Zagieboylo, Charles Sherk, Gookwon Edward Suh, and Andrew C. Myers. PDL: A high-level hardware design language for pipelined processors. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 719–732, New York, NY, USA, June 2022. Association for Computing Machinery.

A Literature Survey Statistics

We have surveyed the following conferences:

- International Symposium on Formal Methods (FM)
- International Conference on Computer Aided Verification (CAV)
- International Symposium on Principles of Programming Languages (POPL)
- International Conference on Programming Languages Design and Implementation (PLDI)
- NASA International Symp. on Formal Methods (NFM)
- International Conference on Interactive Theorem Proving (ITP)
- International Conference on Certified Programs and Proofs (CPP)
- International Conference on Formal Methods in Computer-Aided Design (FMCAD)
- International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)
- International Workshop on Instruction Set Architecture (SpISA 2019)

We did not count the total number of publications, but with an estimate of 50 papers per conference, for nine conferences and ten years we get a total of roughly 4500 papers which form the basis of our survey (plus the once-off SpISA 2019 workshop with ten papers).

From these, we selected 120 papers for closer inspection (going by their title and abstract). These form the basis of our survey, and most of them are in the bibliography of this report; the bibliography also includes other work as we saw fit. It is of course not complete, in particular there are a number of historical papers which we omitted.

cyberagentur