Semi-automatic verification of ISA security guarantees in the form of universal contracts

Sander Huyghebaert
Vrije Universiteit Brussel

Steven Keuchel
Vrije Universiteit Brussel

Dominique Devriese
Vrije Universiteit Brussel

***Abstract*—Where ISA specifications used to be defined in long prose documents, we have recently seen progress on formal and executable ISA specifications. However, for now, formal specifications provide only a functional specification of the ISA, without specifying the ISA's security guarantees. In this paper, we present a novel, general approach to specify an ISA's security guarantee in a way that (1) can be semi-automatically validated against the ISA semantics, producing a mechanically verifiable proof, (2) supports informal and formal reasoning about security-critical software in the presence of adversarial code. Our approach is based on the use of universal contracts: software contracts that express bounds on the authority of arbitrary untrusted code on the ISA. We semi-automatically verify these contracts against existing ISA semantics implemented in Sail using our Katamaran tool: a verified, semi-automatic separation logic verifier for Sail. For now, in this paper, we can demonstrate our approach for MinimalCaps: a simplified custom-built capability machine ISA. However, we believe our approach has the potential to redefine the formalization of ISA security guarantees and we will sketch our vision and plans.**

***Index Terms*—ISA security, semi-automatic verification, capability machines**

1. Introduction

An instruction set architecture (ISA) is a specification of the syntax and semantics of machine code. It serves as a contract between software and hardware designers. Traditionally ISAs are specified informally in prose in long architecture manuals. These specification are imprecise, omit details, and offer no way to test/verify advertised guarantees which is critical when talking about security features. The recent trend is to provide formal and executable specifications [1, 4, 8, 9, 13, 23] of ISAs for disambiguation, testability, experimentation and formal study. For instance, Sail [1] is a domain-specific language for the specification of ISAs which is accompanied by a tool that can produce emulators, documentation and proof assistant definitions from a Sail specification. Sail has been adopted by the RISC-V Foundation for the official formal specification of RISC-V, and is used for the development of the CHERI extensions [30]. Such formal specifications are bare necessities for formally verifying hardware (processors) and software (compilers, programs written in assembly).

The functional specification of the semantics is not enough. We also need meta-theoretical statements of the guarantees that programmers rely on. These are traditionally also in prose, but they should be made formal as well, so that they can be used for reasoning about security-critical code and validating ISA extensions. Some recent proposals for formalizing ISA security properties have focused on making ISA extensions explicit about side-channel leakage [10, 14]. However, this work specifies nothing about the behavior of instructions that might be added in new versions or concrete instantiations of the ISA, making them unsuitable for validating security of proposed ISA extensions or for reasoning about security-critical code for an unspecified implementation of the ISA. Other work has focused specifically on security guarantees of capability machines (see below) [19] but has remained fundamentally incomplete (protection domain crossings are out of scope).

We propose to formalize ISA security guarantees in the form of universal contracts, which have already been applied for formalizing capability-safety of high-level languages [7, 27, 28] but also assembly languages [12, 26, 29]. Essentially, the idea is to formulate ISA security guarantees as a contract that applies to arbitrary – including untrusted – code. The contract expresses the restrictions that the programming language enforces on untrusted programs. The universal contracts are formalized using separation logic, an extension of Hoare logic that allows reasoning about programs that use shared mutable data structures, such as the heap [24]. Furthermore, separation logic can be used for sequential programs, as well as concurrent programs.

For now, universal contracts have only been formalized and proven for expressing capability safety of simplified capability machine ISAs and this has required significant effort [12, 26, 29]. In this paper, we propose universal contracts as a more general approach that can capture security guarantees of different security primitives. Additionally, we propose Katamaran, a tool that can be used to validate universal contracts against the Sail-implemented operational semantics of ISAs (or their extensions) as implemented in Sail. The tool is based on a compositional separation logic for μSAIL (a core calculus for Sail) which can be used to define a universal contract for the ISA semantics. It semi-automatically verifies such a contract using a form of symbolic execution, based on a limited amount of user input. This input includes contracts for functions used internally to define the operational semantics and manually-proven helper lemmas that can be used to explain non-trivial reasoning steps to Katamaran. The semi-automation is crucial to make our approach scale to realistic ISAs and to facilitate adapting ISA security

proofs when the ISA changes. To increase trustworthiness, Katamaran is implemented in the Coq proof assistant and comes with a mechanically-verified soundness proof and a sound implementation of the underlying Sail program logic based on Iris [15].

In this paper, we report on our work-in-progress development of the approach and a first demonstration of its application to concrete ISAs. Specifically, Katamaran is currently functional (but the soundness proof is unfinished) and we have finished a proof of capability-safety of MinimalCaps (a minimalistic capability machine ISA), publicly available on GitHub [6]. In a next step, we will extend the approach to minimalistic ISAs with Trusted Execution Environments (TEEs), protection rings and virtual memory. We will present our approach by presenting MinimalCaps (Section 2), Katamaran (Section 3) and the formalization and verification of the MinimalCaps security guarantees (Section 4). Finally, in Section 5, we will discuss our plans to apply the approach to more general and realistic ISAs and ultimately gain more confidence in the security of realistic processors and systems.

2. The MinimalCaps Capability Machine

Capability machines are a special type of processor that offer capabilities, which are essentially pointers that carry a range of authority and permissions. An example of a mature capability machine ISA extension (or family of extensions) is CHERI [30]. Conceptually, capabilities are tokens that carry authority to access memory or an object. When capabilities represent software defined authority, like invoking objects or closures, they are referred to as object capabilities. Capabilities can be represented as a quadruple, (p, b, e, a), consisting of the permission of the capability, the begin address, end address and a cursor. The permissions available on the MinimalCaps machine are currently: O, the null permission, R, the read permission and RW, the read and write permission. Fig. 1 shows the range of authority of a capability is $[b, e]$ and the cursor a denotes what memory location the capability is currently pointing at.

We will introduce capability safety using the capability machine we have developed so far for a first case study, called MinimalCaps. It contains a minimal subset of instructions from CHERI-RISC-V [30], including branching, jumping and arithmetic instructions. The possible values that can be stored in general-purpose registers (GPRs) are integers and capabilities. For simplicity, we restrict the MinimalCaps machine's memory to only contain integers, thus capabilities can only reside in registers. Additionally, MinimalCaps does not yet offer a form of object capabilities. We intend to remove both limitations in the near future.

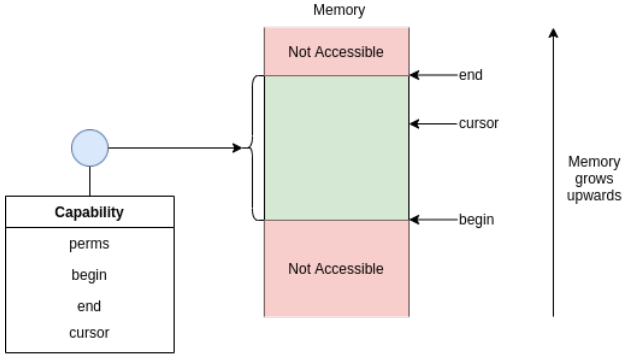

Figure 1. Concept of a capability

The security guarantee we have formulated for our capability machine is capability safety. Our specification of capability safety is based on that of [7, 11, 12, 27, 28]. The capability safety property is defined in terms of the values available on the capability machine. We've split up capability safety into two logical relations, one that is specific to capabilities and one that works on other values as well, such as integers. Fig. 2 shows the logical relation that handles capability only values, $\mathcal{V}_c$, and the logical relation handling all values, $\mathcal{V}$. As one would expect, the case for capability values in $\mathcal{V}$ is defined in terms of $\mathcal{V}_c$.

These logical relations express the authority that is represented by a value or capability, in the form of separation logic predicates that must hold for safely passing it to untrusted code. The definition says that memory capabilities are safe to pass to an adversary when the addressable locations are owned by an invariant that allows arbitrary integers. Note that this definition assumes a form of shared invariants, as available in Iris, indicated by a box. For more expressive capability machines, the definition is complicated further by the presence of object capabilities and the ability to store capabilities in memory, but we refer to existing work for more explanations about that. In terms of these logical relations, the ISA security guarantee (capability safety) states that every instruction will produce safe values in the registers when it is invoked with safe values (see Section 3). By the rules of the program logic, this contract additionally implies that the machine will only use authority that it has access to through the values in the registers.

3. Katamaran

Verifying that security properties are upheld by the semantics is a serious endeavour and currently requires a lot of manual reasoning. For instance, the Coq formalisation of Georges et al.'s [12] capability safety proof for a simple capability machine with 19 instructions requires about 17kLOC. Real ISAs can of course be much larger. Consequently, scaling up verification of ISA properties raises important proof engineering challenges. Furthermore, if the ISA specification changes, i.e. due to updates or entirely new features, or simply for experimentation, the proofs have to be updated as well. For manual proofs, this can result in a significant amount of work.

In a nutshell, proof automation is mission critical for the verification effort to scale reasonably in terms of the

$$\mathcal{V}_c(c) \begin{cases} \mathcal{V}_c(O, -, -, -) & \triangleq \ True \\ \mathcal{V}_c(R,\ b,\ e,\ -) & \triangleq \ *_{a\in[b,e]} \boxed{\exists n, a \mapsto n} \\ \mathcal{V}_c(RW,\ b,\ e,\ -) & \triangleq \ *_{a\in[b,e]} \boxed{\exists n, a \mapsto n} \end{cases}$$

$$\mathcal{V}(w) \begin{cases} \mathcal{V}(z) \triangleq True & z \text{ is an integer} \\ \mathcal{V}(c) \triangleq \mathcal{V}_c(c) & c \text{ is a capability} \end{cases}$$

Figure 2. Logical relations for capability safety

size and complexity of the specification of the instructions set and of the specification of the security guarantee itself, and for proofs to be robust due to changes in the specification.

Proof automation means, that uninteresting or repetitive parts of the proof are dealt with fully automatically by means of a tool, library, script, etc.. Ideally, a human should be able to help steer the automation by providing heuristics, and she should also be able to intervene directly and prove certain cases manually where full automation fails. In other words, we want verification to be semi-automatic.

To this end, we are developing Katamaran [17], our own semi-automatic separation logic verifier. Katamaran works with µSAIL, a new core calculus for Sail, deeply embedded in the Coq proof assistant, offering many of Sail's features. For the time being we perform the translation manually, but in the future we want to scale the language up and compile Sail to µSAIL automatically.

Like Sail, we also leave the definition of memory out of the functional specification and require a (user-provided) runtime system to define what constitutes the machine's memory and provide access to it. This is done in Katamaran with foreign functions, i.e. functions that are only declared with their signatures and are callable from µSAIL code, but are implemented in Coq. Furthermore, µSAIL allows the invocation of lemmas (sometimes referred to as ghost statements), which instruct the verifier to take a non-trivial proof step that is verified separately.

The security properties are specified by means of separation logic-based contracts consisting of pre- and post-conditions for all functions, including foreign ones. For this Katamaran contains its own deeply embedded assertion language.

Verifying that functions adhere to their contracts is done via *preconditioned forward symbolic execution* [2, 3] of the function bodies. During the execution, Katamaran tries to discharge proof obligations automatically and otherwise leaves residual verification conditions for the user. Currently, we require that all spatial, i.e. related to registers and memory, proof obligations are dealt with automatically, potentially with help of the user in terms of ghost statements. The produced residual verification conditions will be in first-order predicate logic which the user can prove with the full proof automation that Coq provides.

A question that arises is in which sense the generated verification conditions are sufficient to verify the function contracts. The user does not have to take the output of the symbolic executor at face value: Katamaran comes with soundness proofs. The structure is depicted in Fig. 3. The contracts of both kinds of functions and the code of

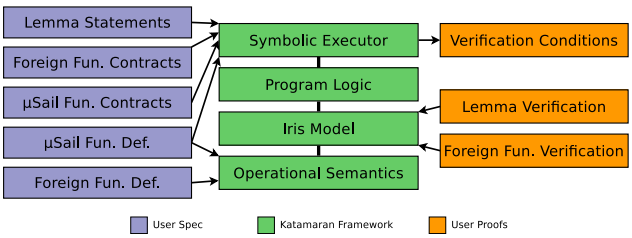

Figure 3. Structure of Katamaran

the µSAIL functions are inputs to the symbolic executor from which it produces verification conditions. A first (work-in-progress) soundness proof connects this to an axiomatic program logic: given a proof of the verification conditions, the function bodies are also verifiable in the program logic.

The program logic consists of separation logic-based Hoare triples. We assign meaning to these triples using the Iris separation logic framework [15] and verify that the triples hold. This requires user-provided proofs that foreign functions adhere to their contracts and that lemmas used in ghost statements are sound. We kept the axiomatic program logic separate from its instantiation using Iris and in theory other logics than Iris can be used. However, we provide the Iris model as the default choice with full (and finished) soundness proofs and hooks for the user to extend it.

A last adequacy proof connects the Iris triples to the operational semantics: every triple that holds semantically is partially correct. For our purposes, partial correctness is sufficient; we assume it is verified separately that the machine cannot get stuck.

4. Verifying MinimalCaps' Security Guarantees

We will now describe our approach for our simplified custom-built capability machine ISA, for which the semantics are specified in Sail. We have performed a manual translation of the Sail specification to µSAIL, which is straightforward and most definitions look identical (without the ghost statements we have added in the µSAIL code). For the remainder of this section we will focus on the machine invariant we have defined for our MinimalCaps machine and verify that it holds.

An example of a function is the $write\_mem$ function, which takes two arguments, a capability and a value to be written to memory. The value will be written to the address denoted by the cursor of the capability. The contract for this figure can be found in Fig. 4, together with the contracts for other functions that we will use in this section. These checks are critical to the capability safety property of the MinimalCaps machine and the machine will go into a failed state when a check fails. The actual write to memory is done by a foreign function, called wM, that takes an address and value to be written to memory. wM is provided by the Sail standard library for the Sail specification and in the runtime system for its µSAIL counterpart.

The contracts for individual instructions require the machine invariant as a precondition and upon succesful execution of the instruction, the machine invariant will still hold. Our program logic contains points-to predicates for registers, $r \mapsto v$, describing a single register, named r, with contents v. The machine invariant is defined over the values of all registers (including the program counter special purpose register) and asserts that the values in these registers are safe:

$$\exists c.pc \mapsto c * \mathcal{V}_c(c) * (\forall r \in GPR.\exists w.r \mapsto w * \mathcal{V}(w))$$

This machine invariant is also upheld by the fetch-decode-execute loop. Note that this statement of capability safety

is simpler than related work [12, 26, 29] because of the lack of object capabilities, but we will strengthen it when we increase the expressiveness of the ISA.

For other internal and external functions the contracts are more specific to what each function does. Consider the contract for the internal function $read\_mem$, which reads the value in memory denoted by the cursor of the given capability. This contract requires that we know that the given capability is safe and after executing the $read\_mem$ function we know that the capability is still safe and that the read value is safe as well:

$$\{\!\{\ \mathcal{V}(c)\ \}\!\}\ read\_mem\ \ c\ \ \{\!\{\ v.\mathcal{V}(v) * \mathcal{V}(c)\ \}\!\}$$

We use the common notation used for Hoare triples for the result value of a statement or function, i.e. v for the result of $read\_mem$.

To give you an idea of how these contracts are verified using Katamaran, Fig. 5 shows the μSAIL implementation of MinimalCaps' store instruction. This instruction takes 3 arguments, a register with the value to be written to memory, a register containing the capability to be used for writing to memory and an immediate value to add to the cursor of the capability (i.e. the contents of hv will be written to $cursor + immediate$, where the cursor is part of the capability in lv). The returned boolean indicates to the fetch-decode-execute loop that the machine should continue executing.

The first two arguments of the store instruction are GPRs and thus their possible values are limited to the available GPRs of the ISA. A new capability c is derived from $base\_cap$ with the immediate added to the cursor and this capability will be used to perform the write to memory of the word w in hv.

Next we use a few lemmas that will modify the pre-condition so that the contract of $write\_mem$ is respected. For simplicity we will assume that $lv = R0, hv = R1$ and ignore the non-relevant parts of the precondition for this discussion.

The first lemma, $specialize\_safe\_to\_cap$, specializes the $\mathcal{V}$ predicate for $base\_cap$ to a $\mathcal{V}_c$ predicate. This is no problem because at this point we are certain that $base\_cap$ is a capability. The $move\_cursor$ lemma will generate a $\mathcal{V}_c$ predicate based on the $base\_cap$ capability for a capability that differs only in the cursor field (the second argument). Remember that capability safety requires that all addresses between $[begin, end]$ are owned by the capability and the values pointed to by these addresses should be safe, it doesn't mention the cursor of the capability.

$$\begin{aligned}
&\{\!\{\ r \mapsto w\ \}\!\}\ read_reg\ \ r\ \{\!\{\ v.\, v = w * r \mapsto w\ \}\!\}\\
&\{\!\{\ r \mapsto w\ \}\!\}\ read_reg_cap\ \ r\ \{\!\{\ c.\, w = c * r \mapsto w\ \}\!\}\\
&\{\!\{\ \mathcal{V}_c(c)\ \}\!\}\ write_mem\ \ c\ \ v\ \{\!\{\ \mathcal{V}_c(c)\ \}\!\}\\
\{\!\{\ pc \mapsto c * \mathcal{V}_c(c)\ \}\!\}\ &update_pc\ \{\!\{\ \exists c.pc \mapsto c * \mathcal{V}_c(c)\ \}\!\}\\
&\{\!\{\ \mathcal{V}(c)\ \}\!\}\ specialize_safe_to_cap\ \ c\ \{\!\{\ \mathcal{V}_c(c)\ \}\!\}\\
&\{\!\{\ \mathcal{V}_c(c)\ \}\!\}\ move_cursor\ \ c\ \ c'\ \{\!\{\ \mathcal{V}_c(c) * \mathcal{V}_c(c')\ \}\!\}\\
&\{\!\{\ \mathcal{V}_c(c)\ \}\!\}\ lift_csafe\ \ c\ \{\!\{\ \mathcal{V}(c)\ \}\!\}
\end{aligned}$$

Figure 4. Contracts for functions and lemmas used in exec_sd (r is used for registers, v and w for values and c for capabilities)

```
{{ (∃ c . pc ↦ c ✳ V_c(c)) ✳ (∀ r ∈ GPR . ∃ w . r ↦ w ✳ V(w)) }}
function exec_sd(hv : GPR, lv : GPR, immediate : int) : bool :=
  let base_cap := call read_reg_cap lv in
  let (perm, beg, end, cursor) := base_cap in
  let c := (perm, beg, end, cursor + immediate) in
  let w := call read_reg hv in
  {{ r_0 ↦ base_cap ✳ V(base_cap) ✳ r_1 ↦ w_1 ✳ V(w_1) ...}}
  use lemma (specialize_safe_to_cap base_cap) ;;
  use lemma (move_cursor base_cap c) ;;
  use lemma (lift_csafe base_cap) ;;
  {{ r_0 ↦ base_cap ✳ V(base_cap) ✳ r_1 ↦ w_1 ✳ V(w_1) ✳ V_c(c) ...}}
  call write_mem c w ;;
  {{ r_0 ↦ base_cap ✳ V(base_cap) ✳ r_1 ↦ w_1 ✳ V(w_1) ✳ V_c(c) ...}}
  call update_pc ;;
  true
{{ (∃ c . pc ↦ c ✳ V_c(c)) ✳ (∀ r ∈ GPR . ∃ w . r ↦ w ✳ V(w)) }}
```

Figure 5. Simplified version of exec_sd implementation with annotations in double curly brackets

The final lemma, $lift\_csafe$, simply lifts a $\mathcal{V}_c$ predicate to a $\mathcal{V}$ predicate. This is required to uphold the machine invariant, which requires a $\mathcal{V}$ predicate for the contents pointed to by the register denoted by lv.

The $write\_mem$ function will check that the cursor of the capability is within bounds and has the write permission. If this is not the case, the machine will go into a failed state for attempting an illegal write operation. The $update\_pc$ function is quite simple and as one would expect utilizes the $move\_cursor$ lemma to generate a $\mathcal{V}_c$ predicate for the updated pc.

Arriving at the end of the implementation of the store instruction we can verify that its contract holds, i.e. the machine invariant is preserved when executing this instruction. By specifying similar contracts for the other instructions, and the fetch-decode-execute loop, we can conclude that the capability safety property holds for the MinimalCaps machine. This is apparent from the machine invariant because all capabilities the machine has access to must respect the capability safety property.

5. Future Work

The MinimalCaps case study demonstrates a working minimal ISA that allows for further experimentation with universal contracts. In the near future we will extend MinimalCaps to be able to hold capabilities in memory, add instructions for capability inspection and modification, as well as object capabilities. For the former two, we expect only slight modifications to be required to verify that the capability safety property still holds, but object capabilities necessarily complicate the statement of capability safety a bit.

Katamaran. Since proof automation is key, we want to lower the proof burden on the user further. We aim to reduce the need to add certain ghost statements/lemmas, which could be mitigated by making Katamaran aware of certain properties of separation logic predicates like $\mathcal{V}$. For instance, for some of the instructions, we currently need to duplicate predicates – via a lemma – because they are consumed by a function call without being produced. For predicates that are persistent [16], this duplication is automatically fine and we intend to make Katamaran

aware of such predicates and take care of the duplication automatically. We also intend to add support for precise predicates [21], which will reduce some of the branching that currently happens in Katamaran.

It is widely recognized that using separating implication (magic wand) in program verification is convenient and can lead to shorter contracts and proofs. Alas, adding the magic wand quickly leads to undecidability [5] and consequently many verifiers, including Katamaran, do not implement support for it. We want to investigate symbolic execution with limited forms of seperating conjunction and implication [18, 22, 25] to benefit from the convenience.

Universal Contracts. The focus in this paper is on the capability safety guarantee of the MinimalCaps machine, formalized with universal contracts. Our aim is to generalize universal contracts so that they are applicable to more realistic ISAs. To this end we will explore three directions: different security primitives, larger ISA sizes and complex semantic features.

The different security primitives we will focus on are capability machines, which we have presented in this paper, a machine with trusted execution environments similar to Sancus [20] and a machine with protection rings and virtual memory. For each of these security primitives we will develop a minimal ISA at first and formalize security guarantees for these minimal ISAs, demonstrating that universal contracts are applicable beyond the setting of capability machines.

We intend to scale up the number of instructions of the ISAs we take under consideration, to bring them closer to the size of realistic ISAs. The ISAs can then no longer be considered minimal and it will be infeasible to manually translate Sail semantics to μSAIL. We will also need to limit the required amount of annotations to a minimum, i.e. we want to reduce the number of ghost functions required, to keep the proof effort focused on the interesting cases. Increasing the ISA sizes will thus demonstrate the viability of our approach to semi-automate the required proofs.

Complex semantic features like concurrency, interrupts or micro-architectural behavior are orthogonal to the size of the ISA. We separate concerns by first focusing on increasing the size of the ISAs in number of instructions and adding these complex semantic features at a later stage. Taking different complex semantic features into account ensures that our approach must be generalized and will not be limited to those complex features we consider. Specifying universal contracts in such a richer semantic setting requires careful consideration. Features like concurrency can impact the formulation of security properties of ISAs but could also create new ways that the property could be broken in the semantics. Fortunately, Katamaran supports different choices for the underlying program logic and the current default choice is based on Iris [15], a powerful framework for higher-order concurrent separation logic, offering features like guarded recursion and atomic invariants which have been developed for reasoning about complex semantic features of high-level programming languages.

Combining these three directions should result in a generalized universal contracts approach, for which we will have demonstrated various interesting security guarantees for ISAs with different security primitives, a vast number of instructions and complex semantic features. Our next step is then to apply our approach to realistic ISAs. A mature capability machine extension like CHERI [30], which has been instantiated for MIPS and RISC-V, is worth exploring and formulating security properties for. An advantage of considering for example CHERI-RISC-V is that a Sail specification has already been developed and is publicly available. Another viable path to take is to consider other publicly available Sail specifications that offer non-capability security primitives, for example the ISA semantics for RISC-V, specified in Sail [1], which has officially been adopted by the RISC-V Foundation.

References

[1] Alasdair Armstrong et al. “ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS”. In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290384.

[2] Roberto Baldoni et al. “A Survey of Symbolic Execution Techniques”. In: *ACM Comput. Surv.* 51.3 (2018).

[3] Josh Berdine, Cristiano Calcagno, and Peter W. O’Hearn. “Symbolic Execution with Separation Logic”. In: *Programming Languages and Systems*. Ed. by Kwangkeun Yi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. ISBN: 978-3-540-32247-4.

[4] Thomas Bourgeat et al. “A Multipurpose Formal RISC-V Specification”. In: (Apr. 2021). arXiv: 2104.00762 [cs]. URL: http://arxiv.org/abs/2104.00762 (visited on 05/18/2021).

[5] Stéphane Demri, Étienne Lozes, and Alessio Mansutti. “The Effects of Adding Reachability Predicates in Propositional Separation Logic”. In: *Foundations of Software Science and Computation Structures*. Ed. by Christel Baier and Ugo Dal Lago. Cham: Springer International Publishing, 2018.

[6] MinimalCaps Developers. *MinimalCaps Case Study*. 2021. URL: https://github.com/skeuchel/katamaran/tree/main/case_study/MinimalCaps.

[7] Dominique Devriese, Lars Birkedal, and Frank Piessens. “Reasoning about Object Capabilities with Logical Relations and Effect Parametricity”. In: *2016 IEEE European Symposium on Security and Privacy (EuroS P)*. 2016, pp. 147–162. DOI: 10.1109/EuroSP.2016.22.

[8] Shaked Flur et al. “Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA”. In: *Principles of Programming Languages*. Association for Computing Machinery, Jan. 2016, pp. 608–621. DOI: 10.1145/2837614.2837615.

[9] Anthony Fox and Magnus O. Myreen. “A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture”. en. In: *Interactive Theorem Proving*. Ed. by Matt Kaufmann and Lawrence C. Paulson. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 243–258. DOI: 10.1007/978-3-642-14052-5_18.

[10] Qian Ge et al. “Time Protection: The Missing OS Abstraction”. In: *EuroSys Conference 2019*. EuroSys ’19. Association for Computing Machinery,

Mar. 2019, pp. 1–17. DOI: 10.1145/3302424.3303976.

[11] Aïna Linn Georges et al. "Cap' ou pas cap' ?: Preuve de programmes pour une machine à capacités en présence de code inconnu". French. In: *Journées Francophones des Langages Applicatifs 2021*. Institut de Recherche en Informatique Fondamentale, Apr. 2021.

[12] Aïna Linn Georges et al. "Efficient and Provable Local Capability Revocation Using Uninitialized Capabilities". In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: 10.1145/3434287.

[13] Shilpi Goel, Warren A. Hunt, and Matt Kaufmann. "Engineering a Formal, Executable X86 ISA Simulator for Software Verification". en. In: *Provably Correct Systems*. Ed. by Mike Hinchey, Jonathan P. Bowen, and Ernst-Rüdiger Olderog. NASA Monographs in Systems and Software Engineering. Cham: Springer International Publishing, 2017, pp. 173–209. ISBN: 978-3-319-48628-4. DOI: 10.1007/978-3-319-48628-4_8.

[14] Marco Guarnieri et al. "Hardware/Software Contracts for Secure Speculation". In: *Proceedings of the 42nd IEEE Symposium on Security and Privacy*. S&P 2021. IEEE, 2021.

[15] Ralf Jung et al. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *Journal of Functional Programming* 28 (2018). DOI: 10.1017/S0956796818000151.

[16] Ralf Jung et al. "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning". In: *SIGPLAN Not.* 50.1 (Jan. 2015), pp. 637–650. ISSN: 0362-1340. DOI: 10.1145/2775051.2676980.

[17] Steven Keuchel, Georgy Lukyanov, and Dominique Devriese. "Katamaran: semi-automated verification of ISA specifications". Extended Abstract. 2020.

[18] Wonyeol Lee and Sungwoo Park. "A Proof System for Separation Logic with Magic Wand". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. ACM, 2014, pp. 477–490. DOI: 10.1145/2535838.2535871. URL: https://doi.org/10.1145/2535838.2535871.

[19] Kyndylan Nienhuis et al. "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)*. 2020, pp. 1003–1020. DOI: 10.1109/SP40000.2020.00055.

[20] Job Noorman et al. "Sancus 2.0: A Low-Cost Security Architecture for IoT Devices". In: *ACM Trans. Priv. Secur.* 20.3 (July 2017). ISSN: 2471-2566. DOI: 10.1145/3079763.

[21] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. "Separation and Information Hiding". In: *ACM Trans. Program. Lang. Syst.* 31.3 (Apr. 2009). ISSN: 0164-0925. DOI: 10.1145/1498926.1498929.

[22] Jens Pagel and Florian Zuleger. "Strong-separation logic". In: *Programming Languages and Systems LNCS 12648* (2021), p. 664.

[23] Alastair Reid. "Who Guards the Guards? Formal Validation of the Arm v8-m Architecture Specification". In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (Oct. 2017), 88:1–88:24. DOI: 10.1145/3133912.

[24] J.C. Reynolds. "Separation logic: a logic for shared mutable data structures". In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

[25] Malte Schwerhoff and Alexander J. Summers. "Lightweight Support for Magic Wands in an Automatic Verifier". In: *ECOOP*. Vol. 37. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 614–638. DOI: 10.4230/LIPIcs.ECOOP.2015.614. URL: http://drops.dagstuhl.de/opus/volltexte/2015/5240.

[26] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. "Reasoning about a machine with local capabilities". In: *European Symposium on Programming*. Springer. 2018, pp. 475–501.

[27] David Swasey, Deepak Garg, and Derek Dreyer. "Robust and Compositional Verification of Object Capability Patterns". In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017). DOI: 10.1145/3133913.

[28] Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. "Linear Capabilities for Fully Abstract Compilation of Separation-Logic-Verified Code". In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019). DOI: 10.1145/3341688.

[29] Thomas Van Strydonck et al. "Proving full-system security properties undermultiple attacker models on capability machines". submitted.

[30] Robert NM Watson et al. *Capability Hardware Enhanced RISC Instructions: Cheri Instruction-Set Architecture (Version 8)*. Tech. rep. University of Cambridge, Computer Laboratory, Oct. 2020.