Formalizing, Verifying and Applying ISA Security Guarantees as Universal Contracts

Sander Huyghebaert
Vrije Universiteit Brussel

Steven Keuchel
Vrije Universiteit Brussel

Coen De Roover
Vrije Universiteit Brussel

Dominique Devriese
KU Leuven

Abstract

Progress has recently been made on specifying instruction set architectures (ISAs) in executable formalisms rather than through prose. However, to date, those formal specifications are limited to the functional aspects of the ISA and do not cover its security guarantees. We present a novel, general method for formally specifying an ISA's security guarantees so that they (1) can be semi-automatically verified to hold for the ISA functional semantics, producing a high-assurance mechanically-verifiable proof, and (2) support informal and formal reasoning about security-critical software in the presence of adversarial code. Our method leverages universal contracts: software contracts that express bounds on the authority of arbitrary untrusted code. Universal contracts can be kept agnostic of software abstractions, and can strike the right balance between requiring sufficient detail for reasoning about software and preserving implementation freedom of ISA designers and CPU implementers. We semi-automatically verify universal contracts against SAIL implementations of ISA semantics using our KATAMARAN tool; a semi-automatic separation logic verifier for SAIL which produces machine-checked proofs for successfully verified contracts. We demonstrate the generality of our method by applying it to two ISAs that offer very different security primitives: (1) MINIMALCAPS: a custom-built capability machine ISA and (2) a (somewhat simplified) version of RISC-V with PMP. We verify a femtokernel using the security guarantee we've formalized for RISC-V with PMP. For now, we focus on direct channels and integrity guarantees but we explain how the method can be extended to other guarantees in the future.

1 Introduction

An instruction set architecture (ISA) is a contract between software and hardware designers, defining the syntax, semantics, and properties of machine code. Architecture manuals have traditionally specified the ISA informally through prose. Such ISA specifications can be imprecise, omit details, and offer no means to test or verify advertised guarantees, which is particularly important for the ISA's security features. In support of disambiguation, testability, experimentation, and formal study, a recent trend is to instead use formal and executable ISA specifications [Armstrong et al., 2019, Bourgeat et al., 2021, Dasgupta et al., 2019, Flur et al., 2016, Fox and Myreen, 2010, Goel et al., 2017, Reid, 2017].

For instance, the SAIL [Armstrong et al., 2019] programming language was designed specifically for specifying ISAs. It 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, an open ISA based on established reduced instruction set computing (RISC) principles [Asanović and Patterson, 2014], and is used for the development of the CHERI extensions [Watson et al., 2020]. Furthermore, mature Sail specifications for ARMv8a (mechanically translated from authoritative definitions) and RISC-V are available. Such formal specifications are necessary for formally verifying hardware (processors) and software (compilers, programs written in assembly).

In addition to defining the semantics of their instructions, ISA specifications also make meta-theoretical statements about the guarantees upheld by their instructions. For example, ISAs offering virtual memory typically guarantee that user-mode code can only access memory that is reachable through the page tables. Importantly, such guarantees are not just *descriptive* statements that happen to hold for the current version of the ISA, but *prescriptive* statements which are part of the ISA contract; they must continue to hold for extensions, future versions, and (extended) implementations of the ISA. We consider the formalization of ISA guarantees, in contrast to the traditional prose specification, vital to support reasoning about security-critical code and validating ISA extensions.

In this paper, we are primarily interested in formalizing ISA security guarantees about integrity through direct channels, as a first step towards broader guarantees. In that sense, our work is closely related to recent work on validating security guarantees of capability machine ISAs [Bauereiss et al., 2022, Nienhuis et al., 2020] (see Section 8 for a thorough comparison). Less closely related are recent proposals to make ISAs explicit about side-channel leakage [Ge et al., 2019, Guarnieri et al., 2021], for which no guarantees are offered by current ISA specifications, formal or informal.

Formalizing ISA security guarantees requires balancing requirements of various stakeholders. On the one hand, ISA designers and CPU manufacturers require specifications that are abstract and agnostic of software abstractions. They need to be able to easily validate ISAs and their extensions or updates against the specifications, with maximum assurance. On the other hand, authors of low-level software need specifications that are sufficiently precise for reasoning about the security properties of their code. Ideally, they should be able to combine ISA security guarantees (which restrict the authority of untrusted code) with manual reasoning about security-critical, trusted code to obtain full-system security guarantees.

The main contribution of this paper is a general and tool-supported method for formalizing ISA security guarantees, resulting in specifications that are abstract enough to facilitate validation of extensions and updates of the ISA, but that are still sufficiently precise for reasoning about code. The method is based on so-called *universal contracts (UCs)*, which have already been employed successfully for formalizing capability safety of high-level languages [Devriese et al., 2016, Swasey et al., 2017, Van Strydonck et al., 2019], as well as ISAs [Georges et al., 2021b, Skorstengaard et al., 2018, Van Strydonck et al., 2021]. Universal contracts start from the observation that the ultimate goal of security primitives is to reason about trusted code interacting with untrusted code. Essentially, the idea is to work in a program logic for assembly code and formulate ISA security guarantees as a universal contract: a contract that applies to arbitrary —including untrusted— code. This universal contract expresses the restrictions that the ISA enforces on untrusted programs. The program logic enables combining manually verified contracts for trusted code with the universal contract for untrusted code, and proving properties about the combined program. Of course, it must be verified that ISA instructions actually enforce the security properties expressed by the universal contract.

In the context of ISAs, universal contracts have so far only been used for capability machines, where the capability safety property of custom simplified ISAs has been formalized, proven, and applied as a universal contract [Georges et al., 2021b, Skorstengaard et al., 2018, Van Strydonck et al., 2021]. We propose universal contracts as a general approach to formally capture the guarantees of more general ISA security primitives, starting from existing operational semantics. We demonstrate our approach by formalizing the intended security properties for two quite different security primitives: capability safety of a minimalistic capability machine, and memory protection for a (somewhat simplified) version of RISC-V with the Physical Memory Protection (PMP) extension and synchronous interrupts (*i.e.,* exceptions). We prove that the universal contracts hold for the SAIL-implemented semantics, the official formal semantics in the case of RISC-V, using a semi-automated approach that improves over the more manual efforts employed so far. To achieve this, we provide a tool called KATAMARAN: a semi-automatic separation logic verifier for SAIL that automates most boilerplate reasoning in the proofs, and allows focusing on the more interesting parts. For now, this verification relies on some simplifications, most importantly the use of unbounded integers, and we have not yet automated the translation from SAIL to an internal core calculus µSAIL. However, the verification tool itself is fully verified, so that we obtain high-assurance guarantees in terms of the µSAIL semantics.

To summarize, the contributions of this paper are:

- A general method based on universal contracts for formalizing security guarantees of ISAs w.r.t. the operational semantics of the specification language.
- KATAMARAN: a new semi-automatic tool for verifying separation logic contracts on code in µSAIL (a new core language for SAIL). KATAMARAN supports user-defined abstract predicates, lemma invocations, and heuristics. It also includes an automatic solver for pure verification conditions. It is implemented and verified in Coq, based on a general approach described elsewhere. Successful verifications produce machine-checked proofs in an Iris-based program logic that is itself proven sound against the operational semantics of µSAIL.
- A demonstration of the method for two case studies. The first is a minimal capability machine that is a subset of CHERI-RISC-V [Watson et al., 2020]. The second case study is a (somewhat simplified) version of the official formal SAIL semantics of RISC-V with the PMP extension.
- An evaluation of the required effort to validate a UC security guarantee against the functional semantics of an ISA, based on statistics about our two case studies. We measure the amount of effort required to validate the addition of an extra instruction. To assess the effectiveness of Katamaran's (semi-)automation, we compare the MINIMALCAPS verification against a related but more manual proof in Cerise [Georges et al., 2021b].
- An end-to-end verification demonstrating the usefulness of contracts resulting from our method for reasoning about security-critical code. For this purpose we verify an example RISC-V program (called the Femtokernel) that relies on the PMP security guarantees for securing its internal state. It relies solely on the RISC-V PMP universal contract to reason about the invocation of untrusted code.

The remainder of the paper is structured as follows: Section 2 explains the security primitives used in our case studies,

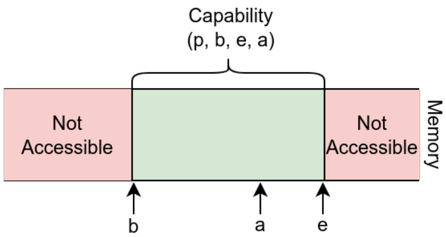


Figure 1: Concept of a capability

while Section 3 presents the general method for formalizing security guarantees of ISAs, *i.e.,* universal contracts. We discuss our new semi-automatic logic verifier, KATAMARAN, in Section 4. In Section 5 and Section 6 we demonstrate the universal contracts method by formalizing and verifying a universal contract for (1) a capability machine and (2) for RISC-V with the PMP extension. We evaluate the effort required for these cases in Section 7, where we also demonstrate the verification of a femtokernel relying on the security guarantee we formalized for the RISC-V PMP case study. Finally, we discuss related work in Section 8 and conclude in Section 9.

2 Background

In this section we cover the security primitives we use in our case studies: capabilities and physical memory protection.

2.1 Capability Machines

Capability machines are a special type of processors that offer capabilities. CHERI is a recent family of capability machine ISA extensions, and includes the Morello ARM extension which is being evaluated in realistic settings by a consortium involving academia and industry [Watson et al., 2020]. Conceptually, capabilities are tokens that carry authority to access memory or an object. When capabilities represent software-defined authority, like applying closures or invoking methods, they are referred to as object capabilities.

Capabilities can be represented as a quadruple, (p,b,e,a), consisting of the permission p of the capability, the begin address b, the end address e, and a cursor a. Permissions on a capability machine can include: the null permission O, the read permission R, and the read and write permission RW. Figure 1 illustrates that the range of authority of a capability is $[b,e]$ and that the cursor a denotes the memory location pointed at by the capability. A special case is the permission E, which models enter capabilities [Carter et al., 1994] [1]. A capability with this permission cannot be used to access memory but can only be jumped to, in which case its permission will change to R. When given to untrusted code, enter capabilities represent a form of encapsulated closures: they can be invoked by the untrusted code, but the caller cannot access the callee's private data and capabilities. As such, they setup a security boundary and can represent a form of software-defined authority and as such they constitute what is generally called an object capability.

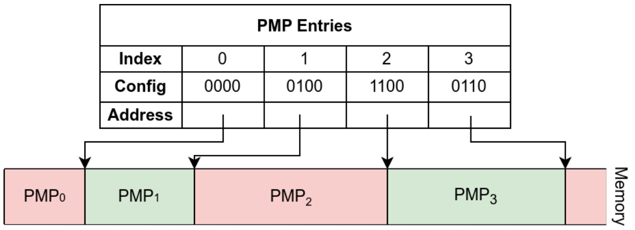
The first case study in this paper is a custom-built capability machine we call MINIMALCAPS. It contains a minimal subset of instructions from CHERI-RISC-V [Watson et al., 2020], including branching, jumping and arithmetic instructions. A word on the machine is either an integer or a capability and these can be stored in memory and general-purpose registers (GPRs). MINIMALCAPS supports memory and object capabilities, a superset of what is supported in Cerise [Georges et al., 2021a,b, Van Strydonck et al., 2021].

2.2 RISC-V PMP

The RISC-V Privileged Architecture Specification provides the optional Physical Memory Protection (PMP) extension to restrict access to physical address regions [RISC-V International, 2022]. RISC-V defines three privilege levels: User, Supervisor and Machine, of which only the Machine level is mandatory for a RISC-V implementation. PMP allows configuring a memory access policy on 16 or 64 contiguous regions of memory by setting special registers, which are only accessible from the most privileged protection level (machine mode). PMP has been used to implement a trusted execution environment called Keystone [Lee et al., 2020].

We illustrate RISC-V PMP policy configuration in Figure 2, where we limit ourselves to four PMP entries. PMP memory regions are specified by a single address register, which is interpreted according to one of several address-matching modes, but for the purpose of presentation, we restrict ourselves to Top of Range (TOR). In TOR mode, the address register of a PMP entry forms the top of the range and the preceding address register (or 0 in case of entry 0) forms the bottom of the range. In other words, for PMP entry i, the range of the entry is defined as $[pmpaddr_{i-1}, pmpaddr_i)$, with $pmpaddr_{-1}$ equal to 0.

In addition to the address, a second special register specifies



PMP Entries				
Index	0	1	2	3
Config	0000	0100	1100	0110
Address				

Figure 2: An example RISC-V PMP policy in Top-of-Range mode (TOR).

[1] Also known as sentry capabilities in the context of CHERI [Watson et al., 2020]

a configuration for every PMP entry. For our purposes we present the configuration as 4 bits of the form $LRWX$, where L defines whether the PMP entry is locked and RWX stands for Read, Write and Execute respectively. We explain PMP using the scenario shown in Figure 2, where we see that we grant read-only access to User and Supervisor mode (U- and S-mode) in PMP entry 1 and read-write access in PMP entry 3. PMP entry 2 contains a configuration where the lock bit is set, indicating that the *read-only* permission of this entry applies not only to U-mode and S-mode, but to M-mode (machine mode) as well. PMP entry 0 grants no permissions and is not locked, so only M-mode can access this range of memory.

We now give a broader explanation of the policy enforced by PMP entries. Non-locked PMP entries grant permissions to U-mode and S-mode. By default, M-mode has full permissions over memory while U-mode and S-mode have no permissions. A locked PMP entry revokes permissions in all modes, thus applying to M-mode. Such a PMP entry can only be reset by resetting the system itself, *i.e.,* one cannot write to the associated configuration and address register (and in the case of TOR addressing mode, the preceding address register).

The PMP check algorithm statically prioritizes the lowest-numbered PMP entries. For a PMP entry to match an address, all bytes (in the case of multi-byte memory accesses) must match the PMP entry address range. When a PMP entry matches an address, the L, R, W, and X bits will determine whether the access succeeds or fails, and if no PMP entry matches an address, the access will succeed in M-mode but fail in other modes.

In our case study we focus on RV32I, the 32-bit base integer instruction set, with the PMP extension. The case itself is a manual translation from the SAIL code to μSAIL, with some additional simplifications: we assume unbounded integers, only two (rather than 16 or 64) PMP configuration entries in top-of-range (TOR) mode are supported, there is no virtual memory, and we only support M-mode and U-mode.

3 Universal Contracts for SAIL ISA semantics

Although the general approach has not been previously described in the literature, universal contracts have been used before to formalize security guarantees for high-level languages [Devriese et al., 2016, Swasey et al., 2017, Van Strydonck et al., 2019] as well as assembly languages [Georges et al., 2021b, Skorstengaard et al., 2018, Van Strydonck et al., 2021]. Essentially, the idea is to formulate the security guarantees offered by a programming language or ISA in the form of a contract that holds for arbitrary, potentially untrusted, code, *i.e.,* a universal contract. The contract expresses the restrictions enforced by the language on untrusted programs and needs to be proven to hold under the language's operational semantics.

Universal contracts are formalized using separation logic, an extension of Hoare logic that enables reasoning about programs that use shared mutable data structures, such as the heap [Reynolds, 2002]. Furthermore, separation logic can be used for sequential and for concurrent programs. The universal contract is a Hoare triple over an arbitrary piece of code, where the precondition and postcondition describe the conditions that need to be met and the guarantees given when executing the code. We will make this more concrete in the case studies discussed in Section 5 and Section 6.

While universal contracts apply to arbitrary assembly code, they take a slightly different form in our setting. When using SAIL, an ISA semantics is defined through a definitional interpreter for the ISA's assembly language, with a main function that implements the fetch-decode-execute cycle of the ISA. The arbitrary programs that our contract applies to therefore take the form of arbitrary instructions encoded in memory. Our universal contract is thus defined as a Hoare triple over the *Fetch-Decode-Execute cycle*. In the remainder of this paper, we will use the function name *fdeCycle* to refer to the SAIL function implementing this cycle, even though it may be named differently in a particular SAIL spec.

The goals of our approach are to enable reasoning about the ISA and programs written in it, as well as preserving freedom of implementation. In other words, we want to define universal contracts for the security guarantees offered by an ISA but leave them sufficiently abstract so that they remain valid under ISA modifications.

Although we verify universal contracts against an ISA's operational semantics, that does not mean it should be considered as derivative or subjugate. Rather, the UC should be regarded as a *security specification*, supplementing the functional specification of the ISA. This security specification is part of the ISA contract and precludes future and current ISA implementations and extensions from adding instructions or other behavior that violate the ISA guarantees, while otherwise preserving their implementation freedom.

We conjecture that the universal contracts defined in the case studies presented later in this paper are sufficiently abstract to allow a range of extensions. For instance, in the case of RISC-V, for which we currently define a contract over the base instruction set and the PMP extension, the universal contract should hold for more optional features of the ISA and extensions (for example adding a floating point unit, user-level interrupts etc.) without requiring major changes to the universal contract. Of course, not all modifications or extensions will be supported by the same universal contract. New semantic features, for example virtual memory, will require modifying the universal contract accordingly.

Ideally, the effort to re-verify a universal contract for a modified ISA with a program verifier is kept minimal. It is commonly understood that a high degree of proof automation leads to proofs that are *robust* to changes [Chlipala, 2013, Pierce et al., 2018]. Therefore, for small modifications or modifications that are orthogonal to security-related matters, we should expect that a semi-automatic verifier, as we present in Section 4, can re-verify proofs with no or only a minimal amount of intervention.

4 Katamaran

Verifying that the semantics upholds security properties is a significant endeavor which involves manual reasoning. For instance, the COQ formalization of Georges et al. [2021a]'s capability safety proof for a simple capability machine with 19 instructions requires about 7kLOC. Real-world ISAs can of course be much larger. Consequently, scaling up verification of ISA properties raises important proof engineering challenges. Furthermore, if the ISA changes (because of minor updates, new features or for experimentation), the proofs have to be updated as well. For manual proofs, this can result in a prohibitive amount of work.

In a nutshell, proof automation is mission-critical for the verification effort to scale in terms of the size and complexity of the specification of the instruction sets and of the specification of the security guarantee itself, and for proofs to be robust to changes in the specification.

Proof automation means that uninteresting or repetitive parts of the proof are dealt with automatically using a tool, library, script etc. The goal is for a human to 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, verifying security properties of ISAs should at least be semi-automatic.

To this end, we have developed KATAMARAN, a new semi-automatic separation logic verifier, implemented and proven sound using Kripke specification monads [Keuchel et al., 2022]. KATAMARAN is developed as a library for the COQ proof assistant, and works with μSAIL, a new core calculus for SAIL, deeply embedded in COQ, offering many of SAIL's features.[2] For the time being, the translation from SAIL to μSAIL has to be performed manually, but we intend to automate it in the future.

Much like SAIL, μSAIL specifications 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 to provide access to it. To this end, KATAMARAN relies on foreign functions – that is, functions implemented in COQ of which the signature has been declared in μSAIL so they are callable. Additionally, μSAIL allows invoking lemmas (sometimes referred to as ghost statements), which instructs 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 postconditions 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* [Baldoni et al., 2018, Berdine et al., 2005] of the function bodies. During the execution, KATAMARAN tries to discharge proof obligations automatically and leaves residual verification conditions for the user where this fails. To bound the burden, we require that all spatial proof obligations – that is, those related to registers and memory, are dealt with during symbolic execution, potentially with the help of the user in terms of ghost statements and heuristics, and thus only pure proof obligations remain. Hence, the produced residual verification conditions will be in first-order predicate logic, which the user can discharge using COQ's built-in proof automation.

A question that arises is whether the generated verification conditions suffice to verify the function contracts. The user does not have to take the output of the symbolic executor at face value: KATAMARAN comes with a full soundness proof against the μSAIL operational semantics. The structure is depicted in Figure 3. The contracts of both kinds of functions and the code of the μSAIL functions are inputs to the symbolic executor from which it produces verification conditions. A first 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 [Jung et al., 2018] 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 principle, other logics than Iris can be used. However, we provide the Iris model as the default choice with full 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.

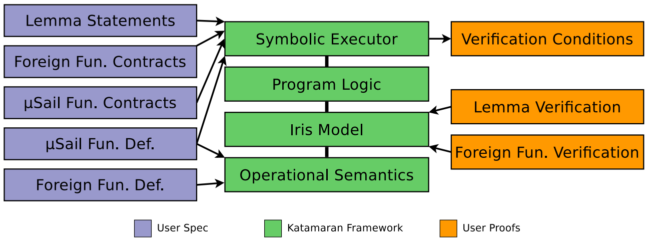


Figure 3: Structure of KATAMARAN

[2] SAIL's existing COQ backend only translates to a shallow embedding.

$$\mathcal{V}(w) \begin{cases} \mathcal{V}(z) & = \quad \mathit{True} \qquad (z \in \mathbb{Z}) \\ \mathcal{V}(O,\ -,\ -,\ -) & = \quad \mathit{True} \\ \mathcal{V}(R,\ b,\ e,\ -) & = \quad \underset{a \in [b,e]}{*} \boxed{\exists w, a \mapsto w * \mathcal{V}(w)} \\ \mathcal{V}(RW,\ b,\ e,\ -) & = \quad \underset{a \in [b,e]}{*} \boxed{\exists w, a \mapsto w * \mathcal{V}(w)} \\ \mathcal{V}(E,\ b,\ e,\ a) & = \quad \triangleright \Box\, \mathcal{E}(R,b,e,a) \end{cases}$$

$$\mathcal{E}(w) = \left\{ \begin{array}{l} \left(\mathrm{pc} \hookrightarrow w * \mathop{\Huge *}_{r \in \mathrm{GPR}} (\exists w.\, r \mapsto w * \mathcal{V}(w)) \right) \twoheadrightarrow* \\ \hfill \mathsf{wp}\, \mathit{fdeCycle}()\ \{\top\} \end{array} \right.$$

Figure 4: Logical relations for capability safety

5 Capability Safety for MinimalCaps

In this section we present the universal contract for and the verification of our first case, MinimalCaps, a subset of CHERI-RISC-V. Capability safety expresses bounds on the authority of arbitrary untrusted code. We prove the capability safety property by defining a contract over the *fdeCycle*, following Devriese et al. [2016], OCPL [Swasey et al., 2017] and Cerise [Georges et al., 2021a,b, Van Strydonck et al., 2021]. The contract states that if we start from a configuration of safe values, arbitrary code will not be able to increase the authority expressed by those safe values.

Figure 4 shows the logical relation $\mathcal{V}$ which defines the authority of words (*i.e.,* integers and capabilities). The logical relation is defined using separation logic [Reynolds, 2002], where $*$ is separating conjunction (unfamiliar readers can interpret it as classical conjunction) and $\mapsto$ the points-to predicate. Points-to assertions $a \mapsto w$ and $r \mapsto w$ represent ownership of the memory location at address a or the register r (respectively) and knowledge of its current contents w. The notation $\underset{a \in [b,e]}{*} P$ indicates that P holds separately for all addresses $a \in [b,e]$.

Authority of a value or capability is defined as separation logic predicates that must hold for safely passing the value or capability to untrusted code. Memory capabilities are thus safe when the addressable locations a are owned by an invariant. This invariant must require exactly that the word stored at address a always remains safe. For simplicity, the definition treats read-only capabilities as read-write. Note that the definition assumes a form of shared invariants, as available in Iris, indicated by a box. The authority represented by an enter capability is software-defined and therefore non-trivial to define. Our definition follows previous work and requires that jumping to the capability with its permission changed to R and the general-purpose registers (GPRs) filled with safe words, will execute correctly and will not break any invariants – that is, $\mathsf{wp}\, \mathit{fdeCycle}()\ \{\top\}$.

First-time readers may ignore Iris's always modality ($\Box$), which requires that the authority does not depend on exclusive ownership of resources, and Iris's later modality ($\triangleright$), which is used to justify the cycle in the definition of $\mathcal{V}$. We refer to prior work for more explanation [e.g., Georges et al., 2021b].

5.1 Universal Contract

The universal contract for MinimalCaps is a contract for the fetch-decode-execute cycle, depicted in Figure 5. We define contracts as Hoare triples, $\{P\}\ \mathit{someCode}()\ \{r.\, Q\}$, where P is the precondition and if $\mathit{someCode}()$ terminates then Q will hold with variable r (the identifier before the ".") bound to the result value. If Q does not mention r, then we omit r and simply write $\{Q\}$. The UC asserts that if the machine is executed (*i.e.,* the *fdeCycle()* is invoked) with authoritized capabilities in pc and general-purpose registers, then it will execute correctly and not break any invariants. The postcondition *True* $(\top)$ is trivial but not very relevant, because the machine will usually continue running indefinitely. It is the contract's implicit guarantees about preservation of invariants that are more interesting for us.

As has been demonstrated previously [Georges et al., 2021a,b, Van Strydonck et al., 2021], such a UC is agnostic of software abstractions but supports reasoning about untrusted code. Essentially, one can register integrity properties of trusted code as invariants (note that Iris invariants can also express protocols on private state [Jung et al., 2018]), and then use the UC for justifying jumps to untrusted code. Applying the UC requires proving that authority is available for all words that the untrusted code gets access to, directly (in a register) or indirectly (in memory reachable from register capabilities). This includes proving that enter capabilities passed to the adversary can be invoked freely but will never break established invariants.

5.2 Verification

The verification of capability safety in the literature so far has required significant manual effort [Georges et al., 2021b, Skorstengaard et al., 2018, Van Strydonck et al., 2021]. In this section, we demonstrate our semi-automatic approach.

The contract for *fdeCycle()* iterates the following contract for *fdeStep()*, a helper function in the semantics which exe-

$$\left\{ \begin{array}{l} (\exists c.\, (pc \mapsto c) * \mathcal{V}(c)) * \\ \quad \mathop{\Large *}_{r \in \mathrm{GPR}} (\exists w.\, r \mapsto w * \mathcal{V}(w)) \end{array} \right\}\ \mathit{fdeCycle}()\ \ \left\{ \top \right\}$$

Figure 5: Universal Contract for Capability Safety for MinimalCaps.

cutes a single FDE cycle.

$$\{(\exists c.\,(pc \mapsto c) * \mathcal{V}(c)) * \mathop{\ast}_{r\in\mathrm{GPR}}(\exists w.\, r \mapsto w * \mathcal{V}(w)) * IH\}$$
$$fdeStep()$$
$$\{(\exists c.\,(pc \mapsto c) * \mathcal{V}(c) \vee \mathcal{E}(c)) * \mathop{\ast}_{r\in\mathrm{GPR}}(\exists w.\, r \mapsto w * \mathcal{V}(w))\}$$

This internal contract requires an induction hypothesis IH:

$$IH := \Box \triangleright (\forall c.\, pc \mapsto c * \mathcal{V}(c) * \mathop{\Large\ast}_{r\in\mathrm{GPR}} (\exists w.\, r \mapsto w * \mathcal{V}(w)) \mathrel{-\!\!*} \mathsf{wp}\, fdeCycle()\, \{\top\})$$

Note how the postcondition allows the pc to contain a safe capability or one that satisfies the expression relation $\mathcal{E}$ above. The latter is necessary because after invoking an enter capability, the pc may contain a value that would not be safe to hand to an adversary but is nevertheless safe to execute. We apply the same contract to helper functions used by *fdeStep*() to execute individual instructions. For other helper functions, the contracts are more specific to what each function does.

Consider, for example, the *read_mem*(c) function, which reads the word in memory denoted by the cursor of the given capability. The contract of *read_mem* requires authority for capability (p,b,e,a) before executing *read_mem*(c) with permission p including at least the read permission, and guarantees that the capability is still safe afterwards, and that authority for the word read from memory is also available:

$$\{\mathcal{V}((p,b,e,a)) * R \leq_p p\}$$
$$read\_mem((p,b,e,a))$$
$$\{w.\, \mathcal{V}(w) * \mathcal{V}((p,b,e,a))\}$$

To give you an idea of how these contracts are verified using KATAMARAN, Figure 6 shows the µSAIL implementation of MINIMALCAPS' store instruction, with verification annotations in red (not part of the code itself), and Figure 7 displays the contracts for the functions used in the implementation. This instruction takes 3 arguments. The first two arguments, *rs* (source contents to write to memory) and *rb* (base capability for computing the target memory address to write to), are GPRs and thus their possible values are limited to the available GPRs of the ISA. The third argument is an integer *immediate*, which is added to the cursor of the base capability (i.e., the contents of *rs* will be written to $cursor + immediate$, where the cursor is part of the capability in *rb*). The returned boolean indicates to the fetch-decode-execute loop that the machine should continue executing.

In the function body, a new capability c is derived from *bc* with the immediate added to the cursor, and this capability is used to perform the write to memory of the word w in *rs*. We use a few lemmas to modify the precondition in order to satisfy the precondition of *write_mem*, which requires authority for the destination capability and for the word being written to memory. For simplicity we assume that $rb = R0, rs = R1$ and ignore the non-relevant parts of the precondition for this discussion.

```
{$(\exists c.\, pc \mapsto c * \mathcal{V}(c)) \mathop{\ast}_{r\in\mathrm{GPR}} (\exists w.\, r \mapsto w * \mathcal{V}(w))$}
store(rs : GPR, rb : GPR, immediate : int) : bool :=
    let bc := call read_reg_cap rb in
    let (perm, beg, end, cursor) := bc in
    let c := (perm, beg, end, cursor + immediate) in
    let p := call write_allowed perm in
    assert p;
    let w := call read_reg rs in
    lemma subperm_not_E RW perm;
    {$r_0 \mapsto bc * \mathcal{V}(bc) * r_1 \mapsto w_1 * \mathcal{V}(w_1) * perm \neq \mathrm{E} \ldots$}
    lemma move_cursor bc c;
    {$r_0 \mapsto bc * \mathcal{V}(bc) * r_1 \mapsto w_1 * \mathcal{V}(w_1) * perm \neq \mathrm{E} * \mathcal{V}(c) * \ldots$}
    call write_mem c w;
    call update_pc;
    true
{$(\exists c.\, pc \mapsto c * \mathcal{V}(c)) * \mathop{\ast}_{r\in\mathrm{GPR}} (\exists w.\, r \mapsto w * \mathcal{V}(w))$}
```

Figure 6: Capability safety for the store instruction.

The *move_cursor* lemma will generate a $\mathcal{V}$ predicate based on the *bc* capability for a capability that differs only in the cursor field (the second argument). Remember that the authority of memory capabilities requires that all addresses between $[begin, end]$ are owned and point to words whose authority is available, *i.e.,* it does not mention the cursor of the capability. Because move_cursor only works for non-E capabilities, we use another lemma subperm_not_E to derive that $perm \neq \mathrm{E}$ from RW $\leq_p$ *perm*.

The *write_mem* function takes two arguments, a capability and a word to be written to memory. *write_mem* will check that the cursor of the capability is within bounds but assumes it has the write permission. If all checks pass, the given word will be written to the address in memory denoted by the cursor of the capability argument. The checks are critical to the capability safety property of the MINIMALCAPS machine and the machine will go into a failed state for attempting an illegal write operation if the checks are not satisfied.

The actual write to memory is performed through a foreign function, called *wM*, which takes an address and a word 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 *update_pc* function is quite simple and, as one might expect, utilizes the *move_cursor* lemma to generate a $\mathcal{V}$ predicate for the updated pc.

Arriving at the end of exec_sd function, we can verify that its contract holds, *i.e.,* safety of register values is preserved when executing this instruction. Together with the verification of other functions, we derive the contract for the fetch-decode-execute cycle. That contract is a universal contract of the ISA, as it expresses an authority boundary on (untrusted) code. It allows us to conclude that our MINIMALCAPS ISA actually satisfies the intended capability safety property.

6 Memory Integrity for RISC-V PMP

In addition to MINIMALCAPS, which is an artificial ISA defined by us, we demonstrate the generality of universal contracts and Katamaran by applying them to RISC-V with support for exceptions (synchronous interrupts) and the PMP extension, as explained in Section 2. The universal contract will capture the memory integrity guarantee offered by the ISA when invoking untrusted code.

Our model of RISC-V is translated to µSAIL from the ISA's canonical SAIL semantics with a number of simplifications and assumptions: we assume unbounded integers, only two (rather than 16 or 64) PMP configuration entries in top-of-range (TOR) mode are supported, there is no virtual memory, and we only support M-mode and U-mode. A WIP version of the model with bounded integers is included in the supplementary material. It includes working versions of the Katamaran-automated proofs, but does not yet have a working Iris model.

We define the universal contract for this machine, over the fetch-decode-execute cycle as shown in Figure 8. In this contract the machine starts from a Normal state, which requires ownership (and knowledge of the current values) of the architectural registers *pc*, *cur_privilege*, *mtvec*, *mcause*, *mstatus* and *mepc*, containing respectively the program counter, current privilege level, configured exception handler address, cause of the last interrupt, and the privilege level and program counter before the last interrupt. Additionally, the state requires ownership of the general-purpose registers and the current PMP configuration entries. Finally and perhaps most importantly, **Normal** requires ownership of PMP_addr_access entries p, an asbtract predicate that represents ownership of the part of memory that is accessible according to the PMP policy entries (discussed further below).

$$
\begin{aligned}
&\left\{\mathcal{V}(c)\right\}\ \mathit{read\_mem}\ c\ \left\{v.\,\mathcal{V}(v)*\mathcal{V}(c)\right\}\\
&\{r\mapsto w\}\,\mathit{read\_reg}\ r\ \{v.\,v=w*r\mapsto w\}\\
&\{r\mapsto w\}\,\mathit{read\_reg\_cap}\ r\ \{c.\,c=w*r\mapsto w\}\\
&\left\{\mathcal{V}(c)*\mathcal{V}(w)\right\}\mathit{write\_mem}\ c\ v\ \left\{\mathcal{V}(c)\right\}\\
&\left\{pc\mapsto c*\mathcal{V}(c)\right\}\mathit{update\_pc}\ \left\{\exists c.pc\mapsto c*\mathcal{V}(c)\right\}\\
&\left\{\mathcal{V}(p,b,e,a)*p\neq \mathrm{E}\right\}\mathit{move\_cursor}\ (p,b,e,a)\ (p,b,e,a')\hookleftarrow\\
&\qquad\qquad\left\{\mathcal{V}(p,b,e,a)*\mathcal{V}(p,b,e,a')\right\}\\
&\left\{\begin{matrix}(p=\mathrm{R}\vee p=\mathrm{RW})\\ \qquad *\,p\leq_p p'\end{matrix}\right\}\mathit{subperm\_not\_E}\ p\ p'\ \{p'\neq \mathrm{E}\}
\end{aligned}
$$

Figure 7: Contracts for functions and lemmas used in exec_sd (r is used for registers, v and w for values and c for capabilities)

$$
\left\{\begin{array}{l}
\mathit{Normal}(l,h,mpp,entries)\\
\triangleright(\mathit{CSR\ Modified}(l,entries)\mathrel{-\!\!}\mathsf{wp}\,\mathit{fdeCycle}()\ \{\top\})\\
\triangleright(\mathit{Trap}(l,h,entries)\mathrel{-\!\!}\mathsf{wp}\,\mathit{fdeCycle}()\ \{\top\})\\
\triangleright(\mathit{Recover}(l,h,mpp,entries)\mathrel{-\!\!}\mathsf{wp}\,\mathit{fdeCycle}()\ \{\top\})
\end{array}\right\}
$$
$$
\mathit{fdeCycle}()\,\{\top\}
$$

$$
\mathit{Normal}(m,h,mpp,entries)=
$$
$$
\begin{cases}
(\exists i.\,pc\mapsto i)*cur\_privilege\mapsto l*mtvec\mapsto h*\\
(\exists c.\,mcause\mapsto c)*mstatus\mapsto[mpp]*\\
mepc\mapsto mepc*PMP\_entries\ entries*\\
\mathop{\Large\ast}\limits_{r\in\mathrm{GPR}}(\exists w.\,r\mapsto w)*PMP\_addr\_access\ entries\ l
\end{cases}
$$

$$
\mathit{Trap}(l,h,entries)=
$$
$$
\begin{cases}
pc\mapsto h*cur\_privilege\mapsto Machine*mtvec\mapsto h*\\
(\exists c.\,mcause\mapsto c)*mstatus\mapsto[l]*\\
(\exists c.\,mepc\mapsto c)*PMP\_entries\ entries*\\
\mathop{\Large\ast}\limits_{r\in\mathrm{GPR}}(\exists w.\,r\mapsto w)*PMP\_addr\_access\ entries\ l
\end{cases}
$$

Figure 8: Universal Contract for Memory Integrity for RISC-V with PMP

Given this authority, the contract states that the ISA will execute correctly, provided that three extra conditions are fulfilled. All three require that the machine continues executing correctly in a specific situation: (1) when CSRs are modified, (2) when a trap occurs to the exception handler, and (3) when an MRET is used to return to a lower privilege level (Recover). The three conditions use the standard IRIS predicate $\mathsf{wp}\,\mathit{fdeCycle}()\ \{\top\}$ which represents the *weakest precondition* for *fdeCycle*() to execute correctly without breaking any invariants. They also use the separating implication operator $\mathrel{-\!\!*}$ (affectionately referred to as the *magic wand*) to require that this weakest precondition holds when authority for the respective premises is presented.

For brevity, we only show the definition of Trap, arguably the most important case, as the other two cases can only be reached if the original privilege level l was Machine, *i.e.,* the UC is used to reason about untrusted Machine code. PMP can be used to encapsulate Machine code (by locking some PMP entries) but it is more typically used for encapsulating lower-privilege code. Trap requires ownership of the same ISA registers and memory as Normal above. However, it additionally requires that the program counter is set to the configured exception handler, that cur_privilege is set to Machine mode, and that mstatus correctly stores l as the previous privilege level. Under these conditions, the user of the UC needs to prove that the machine will execute correctly. This reflects the intuition that trusted code can rely on PMP to encapsulate untrusted lower-privilege code, but only if it ensures security of the configured exception handler.

A crucial predicate in the universal contract is *PMP_addr_access*, which captures the semantics of the PMP check algorithm and is shown in Figure 9. It is defined as a separating conjunction over all addresses of the machine. The predicate allows obtaining a pointsto predicate for an address a, if the PMP policy specifies a permission p (e.g. *Read* or *Write*)for it at privilege level l. Importantly, this means that ownership of other memory locations is not required for using the universal contract.

6.1 Verification

As for MinimalCaps, we verify that our universal contract holds for the functional specification of RISC-V. This verification is done assuming contracts for reading from and writing to memory, shown in Figure 10. The contracts require read or write access, respectively in the form of the *PMP_access* predicate encountered above. We also require that we have ownership of the address that we want to read from or write to, $a \mapsto w$. The postconditions of these functions return the resources used, updated in the case of *write_ram* to point to the newly written value.

Like for MinimalCaps, the proof of the universal contract works by iterating a contract for the single-cycle fdeStep() function. The contract is depicted in a bit more detail in Figure 11. It specifies that executing an instruction will leave the CPU in one of the states Normal, CSRModified, Trap or Recover which we already encountered in the UC, with specific values for the ISA registers. Not shown are the predicates for ownership over the general-purpose registers, *i.e.,* $\ast_{r \in \mathrm{GPR}} (\exists w.\, r \mapsto w)$, and the PMP entries, *PMP_entries*, and *PMP_addr_access*, representing ownership of the PMP-authorized memory. All these predicates are preserved as-is upon a state change, except *PMP_entries* which may be modified in the **CSR Modified** state. The CSRModified and Recover states can only be reached when executing in *Machine* mode, *i.e.,* $p = \mathit{Machine}$. Trap transfers into *Machine* mode and Recover returns to the privilege level stored in mpp.

To verify the memory integrity property for RISC-V with PMP, we define some lemmas that aid in the semi-automatic verification of the case. Figure 12 depicts some interesting lemmas concerning the PMP extension. The first lemma *open_PMP_entries* and a dual lemma called *close_PMP_entries*, open resp. close the *PMP_entries* predicate, to allow direct access to the PMP CSRs in parts of the ISA semantics that access them, particularly the PMP check algorithm. We use the same scheme for reasoning about GPRs, *i.e.,* we pack them in a predicate and open and close it when appropriate. The two lemmas needed for interacting with memory are *extract_PMP_ptsto* and *return_PMP_ptsto*. *extract_PMP_ptsto* trades ownership of PMP-authorized memory, given by *PMP_addr_access* for a pointsto chunk for an authorized, in-range address a and a magic wand that allows to recover *PMP_addr_access* using *return_PMP_ptsto* if we return the pointsto chunk. All these lemmas are proven correct in the Iris model of our case and are explicitly invoked in its function definitions using ghost statements that aid the semi-automatic verification of the contracts of these functions by Katamaran.

$$\begin{array}{l} \mathit{PMP\_addr\_access}\ \mathit{entries}\ m = \\ \quad \ast_{a \in \mathit{addrs}} ((\exists p, \mathit{PMP\_access}\ a\ \mathit{entries}\ m\ p) \mathrel{-\!\!\ast} \exists w, a \mapsto w) \end{array}$$

Figure 9: PMP_addr_access predicate implementation

$$\left\{\begin{array}{l} \mathit{Read} \sqsubseteq t \\ \ast\, \mathit{cur\_privilege} \mapsto p \\ \ast\, \mathit{PMP\_entries}\ \mathit{entries} \\ \ast\, \mathit{PMP\_access}\ a\ \mathit{entries}\ p\ t \\ \ast\, a \mapsto w \end{array}\right\} \mathit{read\_ram}\ a \left\{\begin{array}{l} \mathit{cur\_privilege} \mapsto p \\ \ast\, \mathit{PMP\_entries}\ \mathit{entries} \\ \ast\, a \mapsto w \end{array}\right\}$$

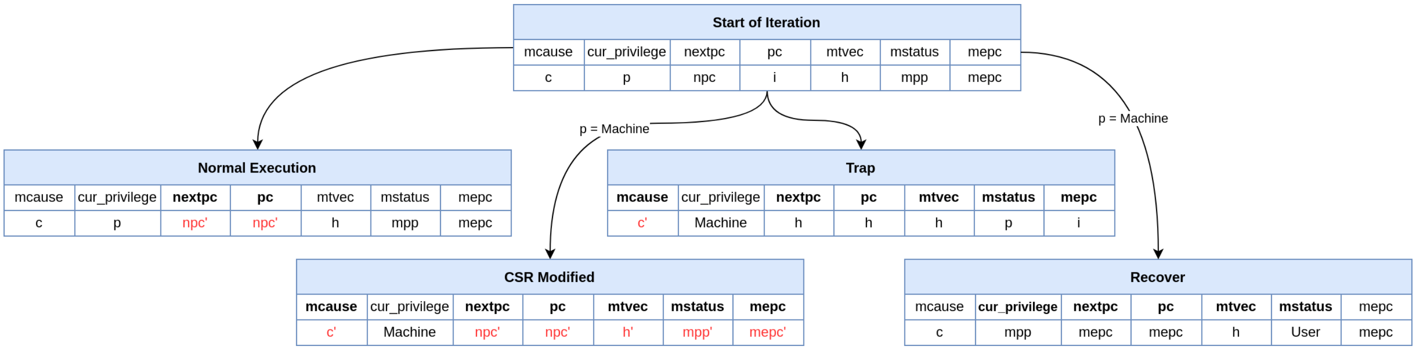
$$\left\{\begin{array}{l} \mathit{Write} \sqsubseteq t \\ \ast\, \mathit{cur\_privilege} \mapsto p \\ \ast\, \mathit{PMP\_entries}\ \mathit{entries} \\ \ast\, \mathit{PMP\_access}\ a\ \mathit{entries}\ p\ t \\ \ast\, \exists w, a \mapsto w \end{array}\right\} \mathit{write\_ram}\ a\ v \left\{\begin{array}{l} \mathit{cur\_privilege} \mapsto p \\ \ast\, \mathit{PMP\_entries}\ \mathit{entries} \\ \ast\, a \mapsto v \end{array}\right\}$$

Figure 10: Contracts for functions interacting directly with memory

These lemma invocations suffice to let Katamaran verify most of the contracts in the codebase. As for MinimalCaps, we only need to prove the contract for fdeCycle() and its lemmas manually, because it requires the use of Iris's later modality and Löb induction, which Katamaran does not (yet) support. The contract expresses PMP's intuitive security guarantee and remains agnostic of software-defined abstractions. Nevertheless, we will see in Section 7.2 that it can be used to reason about specific security-critical code relying on PMP security guarantees.

7 Evaluation

In this section we evaluate our semi-automatic approach to universal contract verification. Our aims for our approach are that universal contracts should be agnostic of software abstractions and verified against the functional semantics of ISAs. Furthermore, we want to minimize the effort to reverify a universal contract for a modified ISA. We evaluate the proof effort required in our case study absolutely as well as relatively to Cerise [Georges et al., 2021a,b, Van Strydonck



Start of Iteration

mcause	cur_privilege	nextpc	pc	mtvec	mstatus	mepc
c	p	npc	i	h	mpp	mepc

Normal Execution

mcause	cur_privilege	**nextpc**	**pc**	mtvec	mstatus	mepc
c	p	npc'	npc'	h	mpp	mepc

Trap

mcause	cur_privilege	**nextpc**	**pc**	**mtvec**	**mstatus**	**mepc**
c'	Machine	h	h	h	p	i

CSR Modified

mcause	cur_privilege	**nextpc**	**pc**	**mtvec**	**mstatus**	**mepc**
c'	Machine	npc'	npc'	h'	mpp'	mepc'

Recover

mcause	**cur_privilege**	**nextpc**	**pc**	**mtvec**	**mstatus**	mepc
c	mpp	mepc	mepc	h	User	mepc

Figure 11: Contract for taking a step on RISC-V (*i.e.,* executing an instruction). New existentially quantified logic variables are shown in red, modified registers are shown in bold. Constraints on the Start of Iteration logic variables are indicated on the arrows (we require for CSR Modified and Recover state transitions that we started from a state running in Machine mode, *i.e.,* $p = Machine$).

$$
\begin{array}{rcl}
\{PMP\_entries\ entries\} & open\_PMP\_entries & \left\{\begin{array}{l} \exists cfg_0,\ addr_0,\ cfg_1,\ addr_1, \\ (pmp_0cfg \mapsto cfg_0 * pmpaddr_0 \mapsto addr_0 * \\ pmp_1cfg \mapsto cfg_1 * pmpaddr_1 \mapsto addr_1 * \\ entries = [(cfg_0, addr_0); (cfg_1, addr_1)]) \end{array}\right\} \\
\left\{\begin{array}{l} PMP\_addr\_access\ entries\ p * \\ 0 \leq addr \leq maxAddr * PMP\_access\ addr\ entries\ p\ acc \end{array}\right\} & extract\_PMP\_ptsto & \{\exists w.\, addr \mapsto w * (addr \mapsto w \multimap PMP\_addr\_access\ entries\ p)\} \\
\{\exists w.\, addr \mapsto w * (addr \mapsto w \multimap PMP\_addr\_access\ entries\ p)\} & return\_PMP\_ptsto & \{PMP\_addr\_access\ entries\ p\}
\end{array}
$$

Figure 12: Contracts for lemmas used in RISC-V PMP case study.

et al., 2021] in Section 7.1. In Section 7.2 we demonstrate how the universal contract can be applied to verify programs running on top of an ISA.

7.1 Proof effort

In Table 1 we present some insightful statistics on our capability safety case study and some relevant statistics for Cerise [Georges et al., 2021a,b, Van Strydonck et al., 2021]. We will first focus on the MINIMALCAPS and RISC-V PMP rows of the table and discuss the comparison with Cerise at the end of this section.

The first column in the table shows the SAIL LoC for the MINIMALCAPS case study. For MINIMALCAPS we started with our own SAIL specification and gradually extended it until it became an actual subset of CHERI-RISC-V, making it trivial to report the SAIL LoC for the MINIMALCAPS case study. We took the opposite direction for the RISC-V PMP case study, starting from the RISC-V SAIL specification and simplifying it during the translation step from SAIL to µSAIL into a minimal subset with the PMP extension. This means we do not have a simplified, minimal RISC-V PMP SAIL codebase and therefore do not report on the SAIL LoC for this case study.

The following column gives the LoC for KATAMARAN, the semi-automatic separation logic verifier used in our approach. KATAMARAN is a reusable tool, both MINIMALCAPS and RISC-V PMP are instantiated for use with KATAMARAN, hence the constant LoC for both case studies. We view both the SAIL LoC and KATAMARAN LoC separate from our case studies and therefore they are not included in the totals at the right of the table.

The next part of the table is data about our case studies themselves. Our case studies are based on SAIL codebases, which we currently manually translate into µSAIL code, but we are confident that this translation can be automated. The µSAIL code is twice the size of the SAIL code, this due to some configuration that we need to provide for KATAMARAN and the required derivation of typeclass instances. Next, we present the number of µSAIL functions, foreign functions, lemmas, and lemma invocations. The lemmas aid KATAMARAN in its verification endeavor and the invocations of these lemmas need to be manually added in the µSAIL functions. The contract proof LoC for the µSAIL specification of our case study are indicative of how well KATAMARAN was able to automate the boring parts. For the MINIMALCAPS case study, the majority of the 122 LoC for the µSAIL specification proofs consists of tactic invocations to discharge trivial proof obligations. This is similar for the RISC-V PMP case study, but KATAMARAN left a few residual verification conditions that required manual discharging. The proofs for the foreign functions and lemmas — the interesting part of our case studies — that reason about capability safety and memory interactions, require manual proof effort. KATAMARAN distinguishes between spatial and pure abstract predicates and provides hooks for a user-defined solver for pure predicates.

			Reusable	Generatable					Specs LoC			Proofs LoC			Model and Solver					
	SAIL LoC	Oper. Sem.	Katamaran LoC	µSAIL LoC	# µSAIL Fns	# Foreign Fns	# Lemmas	# Lemma Invoc.	µSAIL Fns LoC	Foreign Fns LoC	Lemma LoC	µSAIL Fns LoC	Foreign Fns LoC	Lemma LoC	Iris Model LoC	# Pure Pred.	User Solver LoC	Universal Contract LoC	Total LoC	Total LoC (no µSAIL or Oper. Sem.)
MINIMALCAPS	547	-	13k	1117	53	3	12	40	407	49	142	122	194	94	342	3	106	183	2867	1750
RISC-V PMP	-	-	13k	1636	60	3	8	16	851	49	129	212	127	150	144	9	214	272	3880	2244
Cerise	-	1190	-	-	-	-	142	-	-	-	2318	-	-	2918	1351	-	-	-	7919	6729

Table 1: Detailed statistics for the MINIMALCAPS and RISC-V PMP case studies and some comparative statistics (where relevant) with Cerise (the base version without uninitialized, local capabilities or I/O [Georges et al., 2021a]), giving the lines of code (LoC) without comments for different parts of the case study as well as some numbers on how many µSAIL functions, foreign functions, lemmas, lemma invocations and pure predicates each case study defined. There is no direct mapping of our approach with the approach taken for Cerise so the comparison is not entirely fair, for example, the IRIS model LoC for Cerise also contains code for verifying concrete code. We view both the SAIL LoC and KATAMARAN LoC separate from our case studies and therefore they are not included in the totals at the right of the table.

We specified a few pure predicates and report on the LoC for the user solver. Finally, we were able to validate our universal contracts in 183 LoC (*i.e.,* reasoning about the contract for the *fdeCycle*) for MINIMALCAPS and 272 LoC for RISC-V PMP.

We end Table 1 with the total LoC for the case studies, once including the µSAIL specification (or operational semantics for Cerise) and once without, as the µSAIL specification should in principle be generatable.

To further demonstrate the robustness of our approach to universal contracts, we have added an instruction to each case study that doesn't introduce any complexity regarding the proven universal contract, *i.e.,* we are adding a boring case to each case study. We have chosen to duplicate the integer addition instruction for this purpose, which takes three registers, a destination register to write the result to and two source registers. In RISC-V this means adding a new operation for the *RTYPE* instructions, while for MINIMALCAPS we define a completely new instruction. The increase in the µSAIL LoC specification is only two lines for the RISC-V case and 17 for MINIMALCAPS. No further changes are required for the RISC-V case, *i.e.,* we do not need to modify any proofs. For MINIMALCAPS we need to add a lemma invocation in the execute clause for the instruction and we need to specify a contract for the new instruction execution clause, which needs 3 LoC for the contract specification. Furthermore, we add two lines of contract proof code to include this new instruction. Due to the added lemma invocations, KATAMARAN was able to verify the proof without further manual effort. We conclude that adding a boring instruction (*i.e.,* an instruction that is not relevant for the universal contract) requires only minimal changes to both of our case studies.

The categories for which we provide statistics for MINIMALCAPS have no direct mapping for Cerise, so we tried to gather information to the best of our knowledge for a meaningful discussion, but the reader should keep in mind that this comparison is not entirely fair. For example, the IRIS LoC for Cerise does not separate lemmas and proofs that are intended for verifying concrete code from those that are used to prove the universal contract. The operational semantics for the capability machine of Cerise is comparable with our µSAIL specification. More interesting is the statistics for lemmas and the lines of code for the specification and proofs of these lemmas, where Cerise requires significantly more proof effort. For MINIMALCAPS, we use KATAMARAN to automate the boring stuff, leaving us with a smaller amount of proof code needed for the lemmas that are important for the capability safety property. The IRIS model is already partially instantiated in the KATAMARAN codebase, making the MINIMALCAPS LoC for this part smaller than that of Cerise.

7.2 Applying the universal contract: femtokernel verification

Thus far, we have focused on the verification of the security guarantees of our universal contracts. In this section, we demonstrate that the universal contract can also be applied for the verification of properties of programs running on top of an ISA. The MINIMALCAPS UC is close to the Cerise model, for which this has already been demonstrated [Georges et al., 2021a, Van Strydonck et al., 2021]. Therefore, we focus on our RISC-V case, where, to the best of our knowledge, such a verification using universal contracts has not yet been demonstrated.

To illustrate this technique, consider the minimal *femtokernel* in Figure 13, which configures the PMP extension to

```
 1: kernel: la     ra, adv
 2:         csrrw  pmpaddr0, ra, t0
 3:         lui    ra, max
 4:         csrrw  x0, pmpaddr1, ra
 5:         lui    ra, 0x0
 6:         csrrw  x0, pmp0cfg, ra
 7:         lui    ra, 0xf
 8:         csrrw  x0, pmp1cfg, ra
 9:         la     ra, ih
10:         csrrw  x0, mtvec, ra
11:         la     ra, adv
12:         csrrw  x0, mepc, ra
13:         lui    ra, 0x0
14:         csrrw  x0, mstatus, ra
15:         mret
16:
17: ih:     auipc  ra, 0
18:         lw     ra, 12(ra)
19:         mret
20: data:   42
21: adv:    . . .
```

Figure 13: The femtokernel sets up the PMP entries to protect itself, the interrupt handler and its internal state.

protect itself, including its interrupt handler (ih) and a private data field (data), from adversarial user mode code (adv). More specifically, the femtokernel configures the PMP address registers to create the memory regions $[0, \text{adv})$ and $[\text{adv}, max)$ (lines 1–4), where the *max* variable refers to the maximum size of memory available on the machine, then revokes all permissions for user mode for the first region (lines 5–6), and grants read, write and execute permissions to user mode for the second region (lines 7–8). Both entries are unlocked so that machine mode code can also access the first region. The kernel then installs its handler (lines 9–10) and jumps to the adversary in user mode (lines 11–15) by loading the address of the adversary into the *mepc* register (lines 11–12), clearing the *mstatus* register (lines 13–14), i.e. setting the *MPP* field to user mode, and performing the jump (line 15). The handler will read the private data field into the *ra* register before returning, but leaves the value in memory unchanged.

The integrity property we wish to verify is that the private data field, which is initialized with value 42, will always contain the value 42 — that is, code running in user mode cannot modify (or even directly read) the internal state of the kernel. We use the universal contract to reason about the *unknown code* in user mode, and *known code verification* for the initialization and handler code in machine mode.

The contracts for the kernel and interrupt handler are similar, where the integrity property is the essential part. The contracts also require ownership over the GPRs and CSRs, and restricts the PMP entries related CSRs to also remain invariant during execution afer initialization.

Inspired by Islaris [Sammler et al., 2022], we reused existing components and proofs of KATAMARAN to derive a sound verifier for known assembly code and were thus able to verify the contracts for the basic blocks of the femtokernel, *i.e.,* the initialization and handler code.

Taken together, our femtokernel case study demonstrates that our UC for RISC-V can be directly applied for verifying security properties of trusted code relying on PMP to interact with untrusted code. All parts of the verification are fully verified in COQ, yielding a rigorous proof about ISA execution, directly in terms of µSAIL's operational semantics.

8 Related Work

Universal contracts for security were, to the best of our knowledge, first used by Devriese et al. [2016], where they used a reasoning approach based on logical relations in a high-level language, with the fundamental theorem constituting a universal contract. Swasey et al. [2017] used a similar logical relation and universal contract in a logic for Object Capability Patterns (OCP) that enabled them to compositionally specify and verify properties of OCPs in a high-level language. Skorstengaard et al. [2018] use universal contracts to reason about local capabilities in a simple capability machine assembly language, capabilities that temporarily relinquish authority, and a novel calling convention based on them. Similar universal contracts have been formalized and proven for expressing capability safety of simple capability machine ISAs by Georges et al. [2021b], Skorstengaard et al. [2018], Van Strydonck et al. [2019, 2021]. They have taken a verification approach that required significant effort to prove that the universal contracts hold, in contrast to our semi-automatic verification approach enabled by KATAMARAN.

Nienhuis et al. [2020] prove the reachable capability monotonicy (*i.e.,* the authority of available capabilities cannot be increased during normal execution) and intra-domain memory invariant properties for the entire CHERI-MIPS ISA, based on the L3 specification instead of the SAIL specification, where their security property is based on the ISA specification and does not take a hardware implementation or software running on the ISA into account. There are some differences between their work and ours: first, we have demonstrated that the security property we formulate as a universal contract can be used in the verification of programs to be executed on the ISA. Second, the approach taken differs from ours in that they automate the *boring* parts of the proof away with automation using tactics and auto-generated proof scripts, whereas we provide our semi-automatic logic verifier, KATAMARAN, based on symbolic execution. We also argue that a more abstract description of the security property is more appropriate to be future proof against ISA modifications and extensions, and one example where this is beneficial is for registers. In our universal contracts it doesn't matter whether a capability machine has a merged or split register file for capability registers, whereas Nienhuis et al. mention that such a change required refactoring of the properties and proofs in their approach. Finally, we demonstrate the generality of our universal contracts approach by verifying security properties of non-capability machines.

Similar work to that of Nienhuis et al. is done by Bauereiss et al. [2022] on a full-scale industry architecture, Morello, implementing the CHERI extension. To reason about the ISA, a translation from the Arm ASL specification to SAIL occurs first, and from the SAIL specification it is possible to generate code for proof assistants such as Isabelle and COQ. To verify the security properties, the authors define four properties of arbitrary CHERI instruction execution and use that to verify a concrete implementation (*i.e.,* Morello). As mentioned by Bauereiss et al., a limitation for proving stronger properties, such as capability safety, require proof techniques that currently do not scale up to full-scale industry architectures. This is the issue that we are addressing with our proposed universal contract methodology and KATAMARAN to semi-automatically verify universal contracts. Gao and Melham [2021] formally verify the correct execution of the CHERI-instructions and liveness properties for the CHERI-RISC-V ISA. In this work, the authors focus on capabilities and thus leave the RISC-V instructions out of scope. Their approach focuses on a concrete implementation of the ISA, CHERI-Flute [2022], and they manually translate the SAIL specification to SystemVerilog Assertions for the correctness of CHERI-Flute. The focus of their work, however, differs from ours as we focus on properties that are abstract enough to not be tied to a specific ISA implementation.

Guarnieri et al. [2021] propose hardware/software contracts to formalize security guarantees in a minimal ISA setting that takes side-channel attacks into account. A similar approach is taken by Ge et al. [2018], who propose the *augmented ISA (aISA)* as a contract between the hardware and software, adding guarantees about side-channel leakage to the ISA. Both of these proposals address a different problem than we do: while we leave confidentiality guarantees, microarchitectural aspects and side-channel leakage out of scope, they do the same with security boundaries, architectural security primitives and direct-channel protections. In that sense, they are formalizing a different aspect of ISA security guarantees, which should ultimately be combined with direct-channel guarantees like ours to obtain a complete ISA security specification. In our first results that we present in this paper we consider confidentiality guarantees and side-channel attacks out-of-scope but we intend to further explore this in future work and leave the challenge of how to reason about these on the ISA specification as an open problem for the universal contracts approach.

A functional correctness proof of RISC-V PMP for the Rocket Chip implementation was done by Cheang et al. [2020], as a first effort towards verifying the Keystone [Lee et al., 2020] framework. Their verification effort targets the Rocket Chip generated hardware implementation of the RISC-V ISA and is verified using Uclid5.

9 Conclusion

In this work, we have presented universal contracts: a method for capturing security guarantees of ISAs w.r.t. the operational semantics of the specification language. The universal contracts are defined over the *Fetch-Decode-Execute cycle*, resulting in a contract that holds for arbitrary programs running on top of the ISA. We applied this approach to prove security guarantees offered by our two case studies, (1) a minimalistic capability machine for which we have proven that capability safety holds and (2) the RISC-V base integer instruction set with the PMP extension offering memory integrity protection. The verification of our universal contracts happens semi-automatically with KATAMARAN, a tool we have developed for this. Using KATAMARAN we were able to automate the *boring* parts of the verification away and focus on the interesting cases such as interaction with memory. To achieve this, we define spatial lemmas, verified using the IRIS Proof Mode, that we invoke in the µSAIL functions to guide KATAMARAN in its verification effort.

We conclude this paper with a small discussion of some future work. While our work demonstrates the viability of the universal contracts approach, so far we have only instantiated it for cut-down ISAs. A challenge for the universal contracts approach is to apply it to realistic ISAs and take complex semantic features (such as asynchronous interrupts, concurrency etc) and other security properties (e.g. confidentiality) into account. To mitigate that limitation we intend to automate the translation from SAIL to µSAIL, improve KATAMARAN automation and combine it with program logics that support complex semantic features and security properties. Nevertheless, our current results already demonstrate the viability and promise of the general approach to verify ISA security properties using universal contracts and KATAMARAN.

Availability

The artifact containing the Coq development for this paper can be found at https://doi.org/10.5281/zenodo.7188102

References

CTSRD-CHERI/Flute: RISC-V CPU, simple 5-stage in-order pipeline, for low-end applications needing MMUs and some performance., 2022. URL https://github.com/CTSRD-CHERI/Flute.

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. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi: 10.1145/3290384.

Krste Asanović and David A Patterson. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert N. M. Watson, and Peter Sewell. Verified security for the morello capability-enhanced prototype arm architecture. In Ilya Sergey, editor, *Programming Languages and Systems*, pages 174–203, Cham, 2022. Springer International Publishing. ISBN 978-3-030-99336-8.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *Programming Languages and Systems*. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-32247-4.

Thomas Bourgeat, Ian Clester, Andres Erbsen, Samuel Gruetter, Andrew Wright, and Adam Chlipala. A Multipurpose Formal RISC-V Specification. April 2021.

Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware Support for Fast Capability-based Addressing. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 319–327. ACM, 1994. doi: 10.1145/195473.195579.

Kevin Cheang, Cameron Rasmussen, Dayeol Lee, David W Kohlbrenner, Krste Asanovic, and Sanjit A Seshia. Verifying risc-v physical memory protection. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) Workshop on Secure RISC-V Architecture Design*, 2020.

Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013. ISBN 0262026651.

Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. A complete formal semantics of x86-64 user-level instruction set architecture. In *Programming Language Design and Implementation*, pages 1133–1148. ACM, June 2019. doi: 10.1145/3314221.3314601.

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)*, pages 147–162. IEEE, 2016.

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 *Principles of Programming Languages*, pages 608–621. ACM, January 2016. doi: 10.1145/2837614.2837615.

Anthony Fox and Magnus O. Myreen. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In *Interactive Theorem Proving*, Lecture Notes in Computer Science, pages 243–258. Springer Berlin Heidelberg, 2010. doi: 10.1007/978-3-642-14052-5_18.

Dapeng Gao and Tom Melham. End-to-end formal verification of a risc-v processor extended with capability pointers. In *2021 Formal Methods in Computer Aided Design (FMCAD)*, pages 24–33. IEEE, 2021.

Qian Ge, Yuval Yarom, and Gernot Heiser. No security without time protection: We need a new hardware-software contract. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, pages 1–9, 2018.

Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time Protection: The Missing OS Abstraction. In *EuroSys Conference 2019*, EuroSys '19, pages 1–17. ACM, March 2019. doi: 10.1145/3302424.3303976.

Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. Cap' ou pas cap' ?: Preuve de programmes pour une machine à capacités en présence de code inconnu. In *Journées Francophones des Langages Applicatifs 2021*. Institut de Recherche en Informatique Fondamentale, April 2021a.

Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. Efficient and provable local capability revocation using uninitialized capabilities. *Proc. ACM Program. Lang.*, 5(POPL):1–30, 2021b.

Shilpi Goel, Warren A. Hunt, and Matt Kaufmann. Engineering a Formal, Executable x86 ISA Simulator for Software Verification. In *Provably Correct Systems*, NASA Monographs in Systems and Software Engineering, pages 173–209. Springer International Publishing, 2017. ISBN 978-3-319-48628-4. doi: 10.1007/978-3-319-48628-4_8.

Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Hardware/software contracts for secure speculation. S&P 2021. IEEE, 2021.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 2018. doi: 10.1017/S0956796818000151.

Steven Keuchel, Sander Huyghebaert, Georgy Lukyanov, and Dominique Devriese. Verified Symbolic Execution with Kripke Specification Monads (and no Meta-Programming). *Proc. ACM Program. Lang.*, 6(ICFP), aug 2022. doi: 10.1145/3547628.

Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

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 *IEEE Symposium on Security and Privacy (SP)*, pages 1003–1020, 2020. doi: 10.1109/SP40000.2020.00055.

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*. Software Foundations series, volume 1. Electronic textbook, 2018.

Alastair Reid. Who guards the guards? formal validation of the Arm v8-m architecture specification. 1(OOPSLA): 88:1–88:24, October 2017. doi: 10.1145/3133912.

John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

RISC-V International. Specifications - risc-v international, 2022. URL https://riscv.org/technical/specifications/. Accessed: 2022-04-30.

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, page 825–840, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392655. doi: 10.1145/3519939.3523434. URL https://doi.org/10.1145/3519939.3523434.

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. Reasoning about a machine with local capabilities. In *European Symposium on Programming*, pages 475–501. Springer, 2018.

David Swasey, Deepak Garg, and Derek Dreyer. Robust and compositional verification of object capability patterns. *Proc. ACM Program. Lang.*, 1(OOPSLA):89–1, 2017.

Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. Linear capabilities for fully abstract compilation of separation-logic-verified code. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, 2019.

Thomas Van Strydonck, Aına Linn Georges, Armaël Guéneau, Alix Trieu, Amin Timany, Frank Piessens, Lars Birkedal, and Dominique Devriese. Proving full-system security properties under multiple attacker models on capability machines, 2021.

Robert NM Watson, Peter G Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Alexandre Joannou, Ben Laurie, A Theodore Markettos, Simon W Moore, Steven J Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability hardware enhanced risc instructions: Cheri instruction-set architecture (version 8). Technical report, University of Cambridge, Computer Laboratory, October 2020.